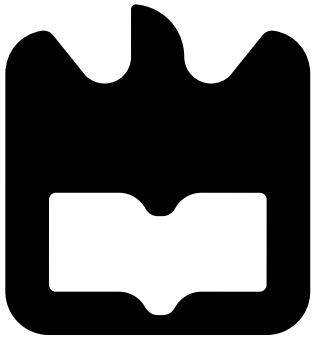
Universidade de Aveiro
Departamento de
Electrónica, Telecomunicações e Informática,
2009

Emanuel Filipe de Ornelas Rodrigues

Software GNU Radio para Detector Digital de Sinais CW com baixa SNR



Emanuel Filipe de Ornelas Rodrigues

Software GNU Radio para Detector Digital de Sinais CW com baixa SNR

Dissertação apresentada à Universidade de Aveiro para cumprimento dos requisitos necessários à obtenção do grau de Mestre em Engenharia Electrónica e de Telecomunicações, realizada sob a orientação científica do professor Dr. Armando Rocha, Professor do Departamento de Electrónica, Telecomunicações e Informática da Universidade de Aveiro

o júri

presidente

João Nuno Matos
Professor da Universidade de Aveiro

vogais

Armando Rocha
Professor da Universidade de Aveiro (orientador)

Adrego Rocha
Professor da Universidade de Aveiro (co-orientador)

Rafael F. S. Caldeirinha
Professor Coordenador; Instituto Politécnico de Leiria; Departamento de Engenharia Electrotécnica

agradecimentos

Gostaria de agradecer ao meu orientador, Armando Rocha, por todo o tempo e paciência que dispendeu ao longo deste ano no apoio à realização desta tese.

Agradeço também ao Adrian David pela ajuda nos primeiros passos dados no sentido da realização deste trabalho.

Não menos importante, fico imensamente grato pelo apoio e camaradagem dentro e fora do ambiente académico de todos os amigos que me acompanharam ao longo desta fase da minha vida.

Finalmente, agradeço à minha família, em especial à minha mãe e irmão, que sempre me apoiaram em todas as ocasiões e possibilitaram que concluí-se esta etapa com êxito.

A todos os referidos os meus maiores agradecimentos.

Palavras-chave: Receptor digital, *Software Defined Radio*, Comunicação por Satélite, Propagação de Ondas, Interferência Atmosférica, Seguimento de Sinal

Resumo

Com a constante evolução das tecnologias de comunicação por satélite, surge a necessidade de expandir a gama de frequências utilizadas quer para melhorar a qualidade dos serviços existentes como disponibilizar novos tipos. Neste âmbito foi proposta uma campanha de medição de um sinal de *beacon* transmitido pelo TDP5 a bordo do AlphaSat. Esta campanha trata-se de um passo essencial para a caracterização de canais de transmissão na banda Q-V em vários pontos da Europa. O satélite será lançado em 2012 e irá transmitir um sinal *beacon* a 39.4GHz.

Pelo facto de o desenvolvimento de um receptor implementado inteiramente em *hardware* ser muito dispendioso, propõe-se nesta tese a substituição de parte dos componentes por um sistema SDR. Este será baseado no equipamento USRP e *software* GNU Radio, o qual fará o controlo de um *frontend* que converte a frequência para 10.7MHz, a sintonização, análise e registo do sinal, assim como o seguimento em frequência deste ao longo do tempo. Esta solução apresenta custos mais reduzidos e maior flexibilidade nas funções que pode desempenhar.

São apresentados os métodos de programação de um sistema desta natureza, a interface de utilizador para configuração do receptor e os testes de funcionamento simulando as condições de recepção esperadas.

Key Words: Digital Receptor, Software Defined Radio, Satellite Communications, Wave Propagation, Atmospheric Interference, Signal Locking,

Abstract

As the satellite communication develop the need to expand the available frequency range, in order to improve the existent services and offer new ones, have risen. In this matter a campaign to monitor a beacon signal transmitted by the TDP5 onboard of the AlphaSat was was proposed. This satellite is expected to launch in 2012 and will be transmitting at 39.4GHz.

Due to the fact that the implementation of an hardware-only solution is costly, we hereby present an option to exchange part of the circuitry for an SDR system. It will be based on the USRP and GNU Radio platform, which will control the basic frontend downconversion to 10.7MHz, the signal tuning, analysis and logging, as well as the dynamic tuning to follow frequency drifts. This solution presents lower development costs and increased flexibility in functionality.

It is presented the programming method for such system, the user interface to configure the beacon receptor and tests simulating the expected real life conditions.

Conteúdo

Lista de Figuras

Lista de Acrónimos

ADC Analog to Digital Converter

ALSA Advanced Linux Sound Architecture

AM Amplitude Modulation

ATS-6 Applications Technology Satellite-6

BPSK Binary Phase Shift Keying

CDTFT Complex Discrete Time Fourier Transform

CFT Complex Fourier Transform

CIC Cascaded Integrator-Comb

CNR Carrier to Noise Ratio

CO *Copolar* - Sinal polarizado

CRC Cyclic Redundancy Check

CW Continuous Wave

CX *Crosspolar* - Sinal despolarizado

DAC Digital to Analog Converter

DDC Digital Down Convertion

DTFT Discrete Time Fourier Transform

DUC Digital Up Conversion

EIRP Equivalent Isotropically Radiated Power

ESA European Space Agency

FCM *Fade CounterMeasurements*

FFT Fast Fourier Transform

FIR Finite Impulse Response

FLL Frequency Locked Loop

FM Frequency Modulation

FPGA Field Programmable Gate Array

FT Fourier Transform

FTP File Transfer Protocol

GMSK Gaussian Minimum Shift Keying

GPGPU General Processing Graphics Processing Unit

GPL General Public Licence

GPU Graphics Processing Unit

GRC GNU Radio Companion

GUI Graphical User Interface

I&D Investigação e Desenvolvimento

IC Integrated Circuit

ICDTFT Inverse Complex Discrete Time Fourier Transform

IF Intermediate Frequency

IIR Infinite Impulse Response

IRDTFT Inverse Real Discrete Time Fourier Transform

LNA Low Noise Amplifier

NASA National Aeronautics and Space Administration

NCO Numerically Controlled Oscillator

NSD Noise Power Spectral Density

OL Oscilador Local

OSS Open Sound System

PGA Programmable Gate Amplifier

PLL Phase Locked Loop

QAM Quadrature and Amplitude Modulation

QoS Quality of Service

QPSK Quadrature Phase Shift Keying

RDTFT Real Discrete Time Fourier Transform

RF Radio Frequency

RFT Real Fourier Transform

ROM Read Only Memory

RX Recepção

SDR Software Defined Radio

SO Sistema Operativo

SPI Serial Peripheral Interface

SWIG Simplified Wrapper and Interface Generator

TCP Transmission Control Protocol

TDP Technological Demonstration Payloads

TEC Total Electron Content

TX Transmissão

UDP User Datagram Protocol

UHF Ultra High Frequency

USB Universal Serial Bus

USRP Universal Software Radio Peripheral

VCO Voltage Controlled Oscillator

VHF Very High Frequency

Capítulo 1

Introdução

1.1 Enquadramento

Uma das maiores riquezas geradas pelo ser humano é a informação. Por esse motivo, desde sempre foram investigados e desenvolvidos métodos de transmitir esse bem. Desde a simples linguagem, passando por cartas, linhas de telefone e culminando na tecnologia de satélites actual. Hoje mais do que nunca, grandes quantidades de informação são geradas todos os dias. Surge por isso a necessidade de estudar métodos mais eficazes de comunicação.

Como foi mencionado, nos últimos anos uma das tecnologias mais investigadas é a comunicação rádio, em especial com recurso a satélites. Estes permitem o envio e recepção de dados através de ondas electromagnéticas a distâncias que uma simples antena terrestre não seria capaz de alcançar devido à curvatura da Terra e outro tipo de obstáculos além de oferecer cobertura a comunidades sem quaisquer infra-estruturas de comunicações. Estações localizadas na superfície terrestre enviam dados através de uma antena para o satélite, e este reencaminha-os para a estação destinatária que pode estar localizada a milhares de quilómetros na face oposta do globo. De modo a melhorar a qualidade deste tipo de ligação, os sistemas actuais são capazes de efectuar processamento de sinal e gestão de recursos de forma dinâmica. Esta gestão é feita utilizando uma base de dados global que descreve os efeitos de determinado fenómeno climatérico e os relaciona com a geometria da ligação na transmissão que está em curso.

Devido ao congestionamento do espectro electromagnético, novas gamas de frequência necessitam de ser estudadas para que seja possível aumentar a banda de transmissão disponível. É conhecido o facto de que o aumento da frequência de transmissão implica um acréscimo da influência de factores naturais na interferência observada no sinal de comunicação [29, 41, 43]. Por este motivo são efectuados estudos prévios ao lançamento de novos satélites, para efectuar o mapeamento das condições de propagação em cada ponto do globo. As medições obtidas neste tipo de campanhas permitem o melhoramento das técnicas utilizadas para mitigar os efeitos da atmosfera sobre a transmissão de ondas rádio.

Neste contexto será lançado o satélite AlphaSat [7]. Possui uma órbita geoestacionária, que providenciará cobertura para a Europa, Médio Oriente, África e algumas zonas da Ásia. Entre os seus vários objectivos encontra-se uma campanha de monitorização de transmissão na banda Q-V. Com esta experiência pretende-se obter informação sobre a viabilidade da utilização da gama para fins comerciais. Um dos grandes objectivos será fazer experiências de diversidade espacial com receptores sincronizados distribuídos por toda a Europa. Para efectuar esta operação é necessário a implementação dos receptores e software de aquisição de dados, para que sejam registados e correlacionados também os fenómenos meteorológicos com o nível de sinal observado.

1.2 Objectivos

O objectivo desta tese visa o desenvolvimento de um detector de baixo custo, com capacidade de seguimento de frequência tendo em vista uma possível participação nesta campanha de medidas. Uma contribuição foi igualmente dada para outro trabalho a decorrer respeitante ao desenvolvimento de hardware para o receptor a montante do detector.

A solução proposta baseia-se numa tecnologia emergente, *software defined radio* (SDR), recorrendo à utilização da plataforma GNU Radio [11] em conjunto com o Universal Software Radio Peripheral (USRP) [8]. Com estas duas ferramentas, pretende-se criar uma interface de software de controlo e arquivamento entre o receptor e a informação que este adquire ao longo da campanha mencionada.

O sistema efectua a monitorização de dois canais de rádio frequência (RF) centrados a 10.7MHz: o sinal *copolar*(CO) e a sua versão despolarizada *crosspolar* (CX). Realiza a estimativa das suas amplitudes e potência assim como da fase relativa entre o sinal *copolar* e *crosspolar*. Isto é feito com um erro nunca superior a 0.5dB para uma relação sinal ruído (CNR) superior a 27dB/Hz e a 3dB para uma CNR de 17dB/Hz assumindo uma largura de banda de 50Hz. Paralelamente estima a CNR, densidade espectral de ruído (NSD), variância da amplitude estimada, frequência central e regista todos os resultados num ficheiro binário compatível com o software MATLAB [16]. Os resultados são mostrados em tempo real, gráfica e numericamente, numa interface desenvolvida de raiz para o efeito. É dado ao utilizador através desta mesma interface a possibilidade de alterar os parâmetros de seguimento e cálculo de estimativas.

Esta abordagem permite converter problemas de hardware na detecção deste tipo de sinais em algoritmos facilmente reconfiguráveis. Diminui-se assim o tempo e os custos de prototipagem do receptor. Desta forma é possível criar um sistema dinâmico e altamente reconfigurável de forma inteligente em tempo real. Esta é uma característica fora do alcance de sistemas de detecção por hardware a um custo tão reduzido e de baixa complexidade de produção.

Foram ainda exploradas as funcionalidades de interface do sistema rádio digital, nomeadamente uma interface SPI para configurar uma unidade de frequência intermédia (IF) desenvolvida para condicionamento do sinal de entrada da USRP.

1.3 Condições

O sinal de onda contínua (CW) transmitido pelo AlphaSat está centrado em 39.4GHz e a sua potência contida numa largura de aproximadamente 50Hz. Estima-se que venha a ser recebido com uma CNR máxima de 55dB/Hz. Não havendo sistemas digitais para esta gama de frequências de portadora, é necessário um *frontend* RF para converter a frequência do sinal para uma frequência intermédia (IF) na gama de funcionamento do equipamento de amostragem.

A frequência irá apresentar alguns desvios ao longo do dia, sendo necessário ajustar a frequência central de aquisição devido a estreita largura de banda de filtros passa-baixo por software que são implementados. Por outro lado o facto de se poder controlar por software uma unidade de IF e nomeadamente o seu Oscilador Local (OL) torna possível compensar desvios de frequência de longo tempo.

Os principais efeitos observados no sinal recebido são atenuação, despolarização e cintilação e os sinais são arquivados 24 horas por dia a taxas entre uma a dez amostras por segundo. São necessários carimbos temporais para que possam ser correlacionados com os fenómenos meteorológicos que ocorreram no momento da aquisição.

Todo o processamento tem que ser efectuado em tempo real e por isso a capacidade de processamento do computador tem que ser tomada em conta na criação do algoritmo de análise e seguimento de frequência.

Ao longo desta dissertação são explicadas as decisões feitas ao nível de *software* e *hardware* para que todas as condições consideradas e a aquisição seja efectuada correctamente e com o máximo de agilidade possível cumprindo os requisitos propostos.

1.4 Estrutura da tese

O capítulo 2 apresenta uma introdução à comunicação por satélite, onde são abordados vários estudos efectuados para determinar os problemas deste tipo de comunicação. Entre os quais destacam-se a atenuação, cintilação e despolarização. É descrita a missão do satélite AlphaSat, na qual está integrada esta tese e fala-se sobre as características de um receptor para este tipo de experiência.

No capítulo 3 são explicados os princípios fundamentais de análise e processamento de sinal digital. Aqui são focados os conceitos de amostragem, análise no tempo e frequência e algoritmos para estimação de potência, frequência e outros tópicos de interesse para o *software*.

Seguidamente, no capítulo 4 é introduzida a tecnologia de *software defined radio* que serve de base à implementação do detector. São discutidas as vantagens e desvantagens de um sistema desta natureza, o *hardware* e *software* considerado e escolhido para a realização do detector.

No capítulo 5 é descrita toda a implementação abordando todos os pormenores observados e aplicados para que o sistema funcione correctamente e cumpra os objectivos propostos.

No capítulo 6 são apresentados e discutidos os resultados obtidos nos testes finais do sistema de detecção.

Finalmente no capítulo 7 são tiradas as conclusões finais, avaliando o desempenho do sistema em função do esperado e apontando melhoramentos que podem ser aplicados no futuro.

Capítulo 2

Comunicações por Satélite

2.1 Introdução

A comunicação por satélite é baseada na transmissão de dados através de ondas electromagnéticas entre este e uma base situada na superfície terrestre. Os dados são codificados de alguma forma, seja ela FM ou AM em termos analógicos ou BPSK, QAM, *et cetera* para o caso de modulações digitais. Este sinal codificado é misturado com uma portadora e é radiado com uma determinada polarização, linear ou circular, de ou para uma estação em Terra. Sendo o sinal transmitido em meio livre não é possível controlar as condições do canal de transmissão. Isto significa que o sinal sofre alterações causadas por factores externos ao longo do seu trajecto. Esta consequência é inevitável mas pode ser compensada recorrendo a técnicas de gestão de qualidade de serviço (QoS), desde redução da taxa de transmissão a um simples aumento de potência do sinal. Estas técnicas são denominadas por *fade countermeasurements* (FCM) e são tanto mais necessárias quanto maior a frequência.

Devido o aumento constante do número de serviços que utilizam este sistema, o espectro electromagnético torna-se cada vez mais congestionado, o que dá origem a uma maior interferência entre canais adjacentes. Surge então a necessidade de aumentar a frequência da portadora de modo a alargar a gama de frequências disponíveis para transmissão assim como a largura de banda de cada canal para conseguir uma maior capacidade.

O problema que se levanta prende-se com o facto da propagação de uma onda electromagnética ser afectada de maneira diferente para diferentes frequências [41]. Entre os efeitos observados apenas três são causadores de grandes incómodos na detecção, recepção e descodificação do sinal: atenuação, cintilação e despolarização. Estudos efectuados mostram que os fenómenos atmosféricos são os principais culpados pela adulteração do sinal [41, 29, 43] a frequências acima de 3GHz.

Para que as técnicas de gestão da QoS sejam aplicadas são necessárias bases de dados criadas *a priori* relatando os efeitos observados para um sinal a uma dada frequência em função de determinado fenómeno atmosférico e geometria de ligação. Olhando aos artigos mencionados conclui-se que não é possível a reutilização de dados anteriores para avaliar e modelar sistemas de comunicação com novos satélites a funcionar em gamas de frequência diferentes das que foram estudadas. É então necessário criar novas bases de dados de forma a mapear os efeitos provocados pela transmissão na nova gama de frequências.

Os satélites são apenas metade do sistema de comunicação. Estes apenas tem utilidade se existir um destinatário para a informação que estão a gerar ou reencaminhar. O destinatário trata-se de um aparelho que consegue filtrar de todo o espectro apenas a porção que lhe diz respeito. Este tópico será analisado mais detalhadamente em 2.4.

2.2 Efeitos de propagação: atenuação, cintilação e despolarização

Desde o lançamento dos primeiros satélites, Sputnik 1 e 2 em 1957, que se efectuam estudos sobre a propagação de ondas electromagnéticas na atmosfera terrestre. De facto durante o lançamento do Sputnik 1 foram feitas as primeiras medições práticas para medir o efeito sofrido pelas ondas rádio ao atravessar a ionosfera com recurso a um pequeno transmissor RF incluído no satélite.

Com o lançamento do ATS-6, efectuado pela NASA em 1974, surgiu a oportunidade de realizar vários estudos mais aprofundados sobre a influência da atmosfera terrestre em sinais de rádio frequência transmitidos por satélites. O ATS-6 levou consigo *beacons* a 40, 140 e 360MHz. Um desses estudos é apresentado em 1980 por Kenneth Davies [30]. Nele são analisados os efeitos no sinal devido à rotação de Faraday, afectada por ciclos diurnos e anuais. Também neste é correlacionado o conteúdo total de electrões (TEC) na ionosfera com a potência recebida ao longo do ciclo anual. A presença destas partículas ionizadas influencia a trajectória e energia do sinal transmitido. São apresentados alguns esquemas da camada ionosférica ilustrando zonas que contribuem para a atenuação do sinal ao longo do ano. Algumas irregularidades de pequena escala são apontadas tanto na ionosfera como na troposfera. Estas irregularidades levam a curtos períodos de flutuação, a denominada cintilação, que podem variar entre 1 segundo e alguns minutos. Olhando para os gráficos apresentados observa-se que os sinais correspondentes a cada uma das 3 frequências referidas se comporta de forma diferente durante estes períodos.

Estudos mais recentes [40, 41, 29, 43] vieram comprovar que para além do aumento da frequência da portadora os factores meteorológicos são os que mais agravam a qualidade da ligação. Uma lista mais extensa dos factores responsáveis pela alteração das características da onda é apresentada em [39]. Podemos subdividi-los em dois grupos: efeitos devido à troposfera e efeitos devido à ionosfera. Estes últimos são praticamente irrelevantes acima de alguns GHz excepto casos pontuais das anomalias equatoriais. Assim restringimo-nos a frequências acima de 3GHz para as quais, e numa trajectória com ângulo de elevação moderado, poderemos identificar basicamente três efeitos principais causados essencialmente na troposfera: atenuação, despolarização e cintilação. As causas são presença de água líquida (nuvens, nevoeiro, chuva), água sólida (cristais de gelo nas nuvens) e irregularidade de pressão/humidade e temperatura devido a turbulências e ainda a absorção devido a gases (oxigénio e vapor de água).

A atenuação é causada pela absorção e espalhamento da onda essencialmente pelas gotas de água líquida numa complexa dependência das características microfísicas destas: forma, tamanho e temperatura. A atenuação é habitualmente modelada pela taxa de precipitação e aumenta sensivelmente com o quadrado da frequência. Num ano típico a atenuação a 20GHz excede 15dB durante 0.01% do tempo ($\approx$1 hora) em Aveiro. Em termos de balanço da ligação, para além das perdas de propagação em meio livre, temos outras fontes: o erro ao direccionar a antena, erros de polarização da antena e perdas na ligação entre a antena e o equipamento de medição.

Denomina-se de despolarização a passagem de parte da energia contida numa polarização linear ou circular para a sua polarização ortogonal: i.e. linear vertical para linear horizontal ou circular direita para circular esquerda. O sinal recebido na polarização de emissão denomina-se *copolar* e o recebido na polarização ortogonal por *crosspolar*. A relação entre os dois chama-se discriminação da polarização cruzada. O fenómeno ocorre no atravessamento de zonas com chuva ou nuvens com cristais de gelo devido às gotas e cristais não serem esféricos e terem uma orientação preferencial devido a efeitos mecânicos durante a sua queda. A atenuação e fase diferencial das componentes do sinal ao longo dos denominados planos principais do meio origina esta transferência de potência entre as duas polarizações que poderá causar interferência num sistema que reuse a polarização.

A cintilação é uma flutuação do sinal em volta do seu valor médio, e deve-se a turbulência atmosférica que modifica de forma aleatória o índice de refracção criando lentes que focam/desfocam a frente de onda no plano da antena. Elevada humidade e temperatura são efeitos determinantes na

intensidade do fenómeno. Quando a geometria da ligação implica baixos ângulos (<10°) e frequências superiores a 10GHz, pode atingir valores de 10dB em algumas ocasiões excepcionais. Isto é ainda mais severo em regiões costeiras onde irregularidades de grande escala na troposfera são mais comuns.

A atenuação é um fenómeno lento – está relacionado com células de chuva a moverem-se – e assim a amplitude do sinal *copolar* e *crosspolar* pode ser registado até uma amostra por segundo enquanto a cintilação é mais rápida e exigirá amostragem entre quatro a dez amostras por segundo.

2.3 AlphaSat

Trata-se de um equipamento contratado pela ESA com lançamento previsto para 2012 a bordo do foguete Ariane 5 [35].

Diversos módulos estão a ser implementados entre os quais o TDP#5. Trata-se de uma antena para transmissão nas bandas Q-V limitadas entre os 33 e os 75GHz. Este módulo tem 3 objectivos principais:

- Telecomunicações: averiguar o desempenho de ligações utilizando as bandas Q-V e testar técnicas de mitigação dos efeitos de atenuação da atmosfera em sistemas de alta capacidade;
- Científico: recolher informação adicional indispensável ao desenvolvimento de futuros satélites e técnicas de adaptação dinâmica nesta gama de frequências;
- Tecnológico: teste de desempenho de novo *hardware* desenvolvido.

De entre os três items, apenas a área científica é do interesse desta tese. Para que se possam realizar experiências de propagação é necessário, além do emissor de sinal de teste, um receptor colocado no local de interesse. Neste caso serão múltiplos pontos espalhados pelo globo já que se pretende efectuar estudos de desigualdades em larga escala.

Um diagrama dos módulos que compõe a missão proposta pelo satélite é apresentado na figura 2.1.

Na banda Q-V que se destina a experiências de propagação o satélite dá um EIRP em torno dos 26dBW. O sinal transmitido trata-se de uma onda contínua a cerca de 40GHz com uma largura espectral prevista de aproximadamente 50Hz. Com base em outras experiências já efectuadas espera-se um deslize diário de frequência nunca superior a 3kHz e flutuações diárias de cerca de 0.5dB em condições normais. A CNR máxima esperada à entrada do receptor situar-se-á em torno dos 55dB/Hz e um mínimo de 27dB de SNR (correspondente ao ruído contido em 50Hz) será a escala aceitável para medir a atenuação com um erro inferior a 0.4dB.

O satélite estará numa órbita geoestacionária mas terá uma inclinação de até 3º o que implicará algum tipo de apontamento programado que se planeia fazer com um motor passo a passo. A medição das duas polarizações exigirá uma antena de cerca de sessenta centímetros de diâmetro para uma gama dinâmica da atenuação de cerca de 25dB e com um transdutor ortogonal de modos para separar o sinal *copolar* do *crosspolar*. Não será de excluir também a necessidade de considerar o efeito de Doppler sobre a frequência recebida.

2.4 Receptores

Em qualquer tipo de sistema de comunicação, electrónico ou não, existe a necessidade de dois componentes básicos: um emissor para transmitir informação a quem estiver interessado e autorizado a recebê-la, e um receptor cuja função é adquirir a informação que é transmitida. No interesse desta tese encontra-se o AlphaSat, que será o nosso emissor e cabe-nos a nós desenvolver o receptor que faz a aquisição do sinal que será enviado para a Terra.

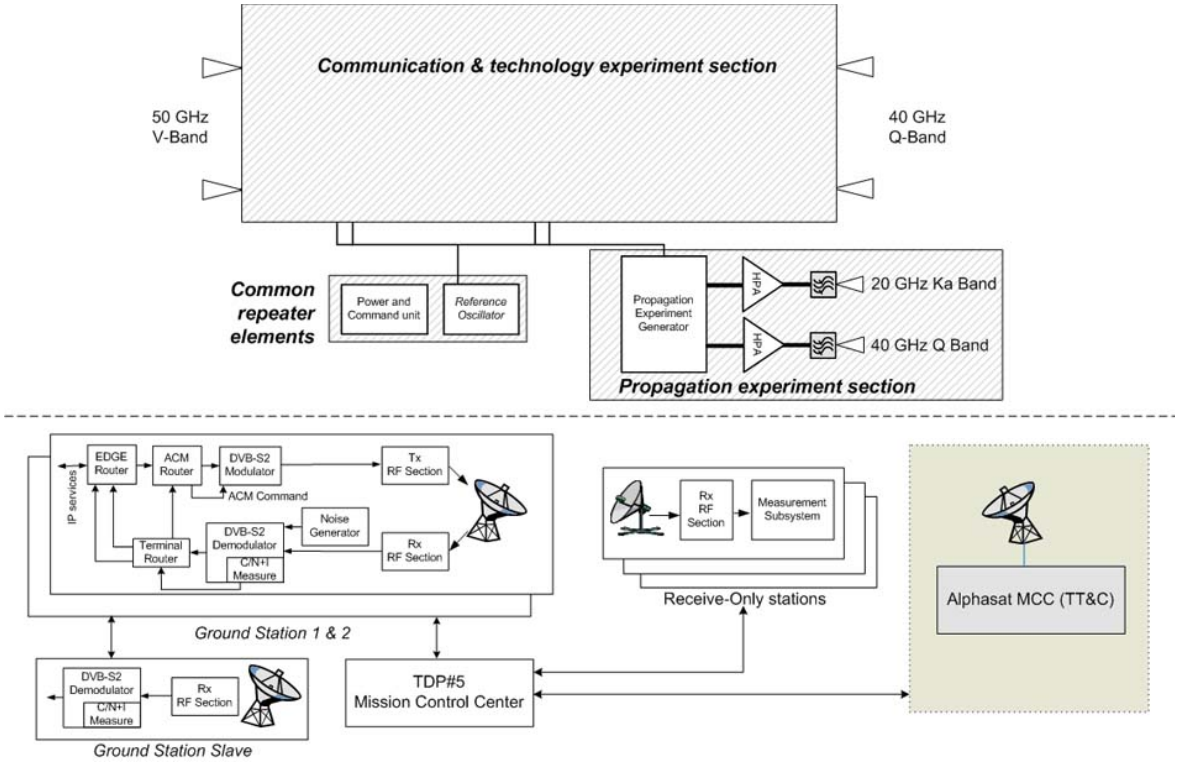


Figura 2.1: Diagrama da missão do AlphaSat

Quando se trata de estudos científicos e testes de telemetria são utilizados *beacons* de satélite que transmitem uma frequência constante (CW) predefinida com uma potência também essa constante. Desta maneira é facilitada a comparação e análise das variações no sinal em estudos de propagação.

Em terra são implementados receptores que sintonizam essa frequência e seguem as suas variações ao longo do tempo. Este acompanhamento da frequência é absolutamente necessário pois o sinal tem que ser estimado na largura de banda mais reduzida possível para evitar a detecção de uma elevada potência de ruído. O acompanhamento consegue-se usando no receptor um oscilador local de frequência controlada por tensão (VCO). Existem duas hipóteses no que respeita ao desenvolvimento destes detectores para receptor de *beacons*: *hardware* e *software*. Ambos têm as suas vantagens e desvantagens, contudo a solução digital para o detector é cada vez mais popular. Existem 2 formas de fazer o seguimento da frequência. A utilização de *frequency locked loop* (FLL) e *phase locked loop* (PLL) é aplicável tanto em *hardware* como em *software*. No entanto, a análise espectral é apenas realizável em *software*.

O problema associado tanto à PLL como à FLL é a sua complexidade e custos de implementação. No caso da PLL existe ainda o risco de perda de sincronismo quando a degradação da CNR é suficientemente grande. Quando isto acontece a tendência é a PLL perder o sincronismo e não o readquirir quando o sinal recupera uma CNR superior aquela em que ocorreu a perda. Os circuitos têm que a auxiliar forçando um varrimento do VCO até que a aquisição natural seja possível o que acontece quando a diferença de frequências entre o sinal e o VCO está dentro da largura de banda da malha.

A maior desvantagem da análise espectral deve-se ao limite de velocidade de processamento do *hardware* digital. Ao longo dos anos, com o surgimento de tecnologias *multicore* este factor tem vindo a ser mitigado, sendo actualmente possível processar grandes quantidades de dados adquiridos em tempo real. A sua maior vantagem baseia-se na flexibilidade da detecção de picos e aplicação de filtros, permitindo a análise uma largura de banda mais elevada para efeitos de *tracking* que a possível

com PLL. Esta mostra-se uma alternativa viável e de baixo custo para implementar o detector de sinal *beacon*.

No que diz respeito à análise espectral, a sua maior desvantagem prende-se com a velocidade do hardware digital que faz o processamento. Devido ao grande desenvolvimento na área de FPGAs e evolução dos processadores para arquitecturas multi-core é cada vez maior a quantidade de informação que se pode processar em tempo real para a implementação deste tipo de sistemas. Uma vantagem clara é a precisão da detecção da frequência. Esta dependente apenas da resolução com que é analisado o espectro, o que permite trabalhar numa largura de banda elevada mantendo a agilidade sem sacrificar a precisão, o que não acontece com uma PLL. Pelo facto de ser um bloco do receptor criado através de software os testes são mais rápidos de executar e os custos envolvidos reduzidos.

2.5 Estado da Arte

Os receptores são habitualmente caros e as cotações obtidas para equipamentos produzidos para duas ou três unidades são exorbitantes. Pretende-se aqui arranjar uma solução integrada que contemple não só o detector propriamente dito mas que ofereça a possibilidade de controlar uma unidade de frequência intermédia como, eventualmente, outras partes do *hardware*.

Recorde-se que o AlphaSat tem por objectivo não só caracterizar o canal num determinado local mas também fazer medidas de diversidade pela Europa o que muito ajudará a avaliar FCM inovadoras e essenciais numa banda em que a atenuação será substancialmente maior que na Ka. Este objectivo de conseguir receptores baratos foi aliás alvo de uma chamada de projecto pela ESA (EMITS).

Os pequenos tamanhos das antenas possibilitam a instalação dos equipamentos em local abrigado o que diminui desde logo as despesas e melhora a fiabilidade.

Enquanto que não se justifica desenvolver a parte de conversão dos 40GHz para uma primeira IF de 1 a 2GHz ou um oscilador a cristal de baixo ruído de fase como referência do receptor, tudo o resto pode ser desenvolvido localmente oferecendo desde logo uma enorme capacidade de intervenção para resolver problemas que possam surgir.

Já foram feitas, anteriormente, experiências em Aveiro com o objectivo de implementar um receptor que permita efectuar uma aquisição tal como a proposta para a missão do AlphaSat.

Um exemplo pode ser visto na tese de mestrado de Ricardo Sousa [31], realizada em 2007. Esta trata uma implementação de um receptor digital para balizas de satélite, podendo ser considerada uma aproximação utilizando SDR. Com recurso a uma DDS e uma DSP, o sinal é digitalizado, processado e enviado para o computador onde é efectuado o controlo e registo de medições.

Com a evolução dos sistemas SDR comerciais, o tempo necessário para testar o *hardware* e aprender os métodos de programação específicos de cada componente digital não compensa a facilidade de aquisição de um sistema completo já existente no mercado. Exemplos como o USRP da Ettus Research custam cerca de 700$ pronto a funcionar, incluindo um conjunto de bibliotecas específicas com uma vasta comunidade de apoio ao desenvolvimento de *software* para esta plataforma. Como complemento, alguns destes sistemas vêm equipados com portas de comunicação muito utilizadas para controlo, tais como I^2C, SPI ou RS232, assim como portas I/O e DACs/ADCs de baixa velocidade. Esta característica permite que se tornem a centralina de um sistema de recepção, sendo capazes, para além de adquirir sinal, de actuar sobre *hardware* externo para efectuar ajustes.

Capítulo 3

Processamento Digital de Sinal

O detector implementado baseia-se em estimação espectral pelo que aqui se introduzem os conceitos mais relevantes. Como se descreverá adiante o hardware transfere para o PC os sinais amostrados na forma das componentes cartesianas. Com elas terá que ser estimada a amplitude dos sinais *copolar* e *crosspolar* com a maior precisão e sensibilidade possível. Aqui introduzem-se os conceitos mais relevantes que ajudarão também a compreender alguns compromissos a tomar tanto mais que o sinal virá acompanhado de ruído branco gaussiano que torna a identificação do sinal mais problemática quando o desvanecimento é profundo.

3.1 Amostragem

Para que o processamento de sinal seja feito de forma digital é necessária a conversão de dados analógicos para o formato digital. Este processo é chamado de digitalização. Considerando a digitalização como uma caixa negra, na entrada temos um sinal contínuo no tempo e em amplitude, e na sua saída um sinal discreto no tempo e em amplitude. O acto de discretizar um sinal no tempo chama-se amostragem, enquanto que a discretização do sinal em amplitude é denominada quantificação. O componente que efectua esta operação é um conversor analógico para digital (ADC).

O processo de amostragem consiste em quantificar o nível de sinal num determinado instante e repetir o processo periodicamente. A frequência de amostragem (f_a) é o número de amostras que são feitas por unidade de tempo. A expressão 3.1 indica como é calculada a frequência de amostragem com base no período esperado entre amostras (T_a).

$$f_a = \frac{1}{T_a} \tag{3.1}$$

Pelo facto de ser impraticável ter uma frequência de amostragem infinita, em que teríamos informação sobre todos os instantes de tempo, a perda de informação do sinal original é inevitável. Intuitivamente chegamos à conclusão de que quanto menor for f_a mais informação será perdida. Um exemplo desta situação é apresentado na figura 3.1. Neste caso temos um sinal sinusoidal com frequência de 1Hz. São feitas 3 amostragem a ritmos diferentes com duração de 2 segundos. Como é fácil de observar, o sinal que apresenta a forma mais próxima de uma sinusóide perfeita é o de maior frequência de amostragem. O primeiro gráfico, representa a amostragem feita a 2Hz, em que apesar da forma de onda estar completamente alterada, ainda é possível recuperar a frequência e amplitude correctamente. Podemos ver no terceiro exemplo da figura 3.1, que embora não seja uma frequência infinita, contém informação suficiente para recuperar a forma do sinal sem qualquer distorção. Vários estudos foram realizados nesta área. O mais importante resultou no teorema de Shannon-Nyquist que será brevemente explicado no tópico 3.1.1. Este teorema estabelece os princípios principais a ter em conta num sistema digital em termos de amostragem e recuperação de sinal.

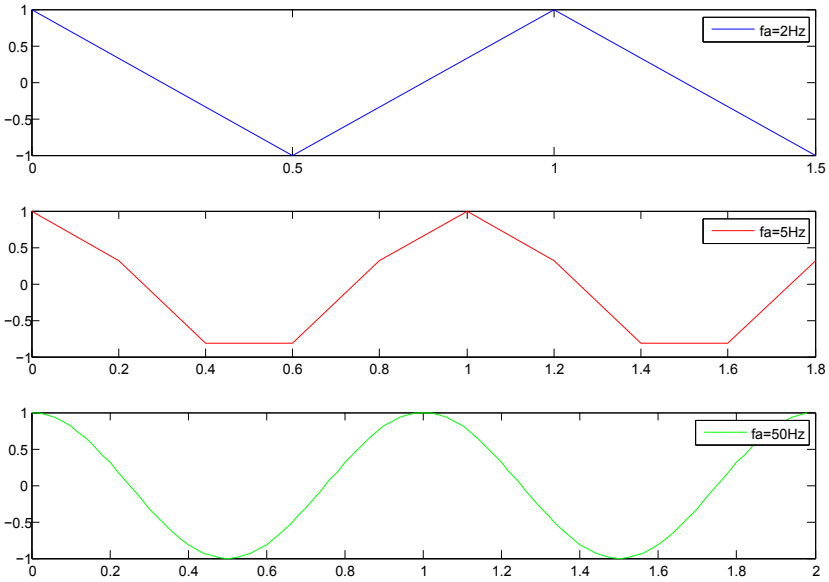


Figura 3.1: Amostragem de sinusóide de 1Hz a 3 frequências diferentes

As *regras* definidas pelo teorema são utilizadas para escolher o ADC que fará a transição do mundo analógico para o digital.

Um ADC tem dois parâmetros importantes no que respeita à digitalização: resolução e frequência de amostragem.

A resolução é dada pela excursão de sinal à entrada do ADC a dividir pelo número de bits do valor gerado na sua saída como indicado pela expressão 3.2.

$$ADC_{res} = \frac{V}{2^{n_{bits}}} \tag{3.2}$$

Deve ser tido em conta que a resolução efectiva do ADC não pode ser medida através desta fórmula. A existência de um número de *bits* menos significativos não exactos é incontornável. É necessário, em todos os casos, verificar as especificações do fabricante de modo a garantir que a exactidão da digitalização efectuada pelo ADC escolhido esteja dentro dos limites desejados. Uma explicação mais detalhada sobre a escolha deste componente pode ser encontrada em [42].

A frequência de amostragem irá limitar a largura de banda do sinal a observar. Quanto maior for a f_a maior será a gama do espectro que poderemos analisar.

3.1.1 Teorema de Shannon

O teorema diz-nos que se uma função $x(t)$ não contém frequências superiores a BHz, então ela pode ser completamente determinada por uma série de pontos espaçados de $1/(2B)$ segundos. Isto significa que é possível recuperar integralmente um sinal se for amostrado a um ritmo pelo menos duas vezes superior ao período da componente sinusoidal de maior frequência que o compõe.

Desta afirmação podemos deduzir os seguintes princípios práticos:

- Conhecendo a máxima frequência do sinal a analisar sabemos o limite inferior para a frequência de amostragem a utilizar, também conhecida de frequência de Nyquist;

- Conhecendo a frequência máxima a que o equipamento faz a amostragem sabemos o limite espectral para o sinal que podemos trabalhar

- No caso de a gama de frequências do sinal ultrapassar $f_a/2$ é necessário garantir que as componentes de frequência mais elevadas sejam desprezáveis. Se não o forem deverá ser utilizado um filtro, passa-baixo ou passa-banda, de modo a reduzir a energia dessas componentes;
- Para que se cumpra os limites do conteúdo espectral eram necessários existir filtros perfeitos (em forma de parede). Isso não acontece na vida real e geralmente algumas componentes de frequência dentro da banda de interesse são atenuadas, assim como o conteúdo energético fora da banda não será perfeitamente nulo;
- Todos os pontos acima têm que ser ajustados pela pessoa encarregada de projectar o sistema, devido a ter que lidar com situações não ideais em que são feitos compromissos de desempenho.

Quando frequências acima de $f_a/2$ têm energia não nula ocorre um fenómeno de *aliasing*. Consiste na sobreposição de algumas partes do espectro correspondente a uma imagem replicada do sinal no processo de amostragem. Na figura 3.2 podemos ver um sinal correctamente amostrado, em que após aplicado um filtro ideal para retirar as imagens (a verde) obtemos uma representação em frequência sem erros. No caso da figura 3.3 a frequência de amostragem é inferior a $f_a/2$ e por esse motivo as imagens encontram-se sobrepostas ao espectro do sinal. Esta sobreposição não pode ser evitada recorrendo a filtros como no primeiro caso.

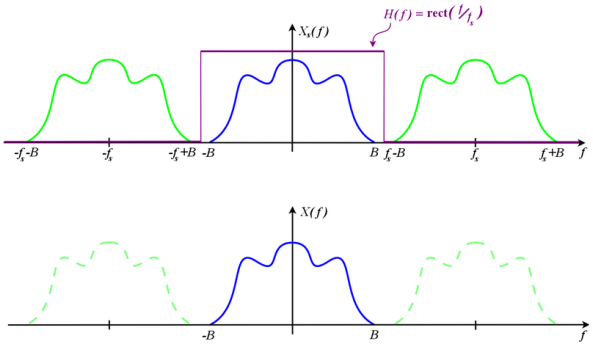


Figura 3.2: Sinal correctamente amostrado - sem *aliasing*

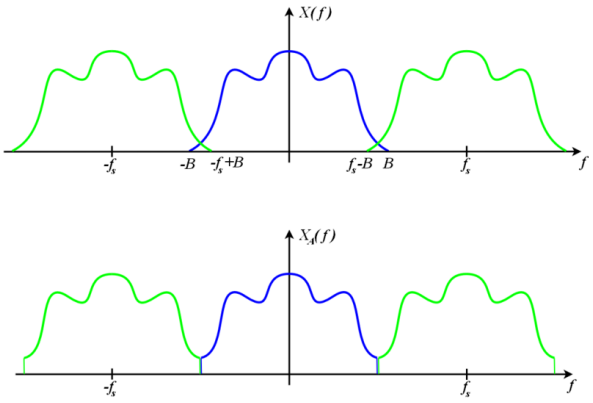


Figura 3.3: Sinal incorrectamente amostrado - com *aliasing*

3.2 Transformada de Fourier

Na área de processamento de sinal a transformada de Fourier (FT) trata-se de uma das ferramentas mais valiosas e poderosas em diversas áreas de processamento de sinal. Consiste no processo

matemático de transformação de uma função no domínio do tempo para o seu equivalente no domínio da frequência. Este método possibilita a separação de diferentes sinais misturados no tempo mas com frequência e fase distintas.

Um exemplo prático é ilustrado na figura 3.4. A situação simula a soma de 2 sinais sinusoidais, x_1 e x_2 a 1Hz com amplitude 2 e 4Hz com amplitude 1 respectivamente. Uma vez somados, a onda resultante não permite distinguir com facilidade ambos os sinais que nela estão contidos. No entanto, ao efectuar a transformada de Fourier é possível identificar claramente na terceira figura os dois sinais e a sua frequência e amplitude correspondentes. A fase também pode ser recuperada fazendo a transformação para a componente real e imaginária em separado.

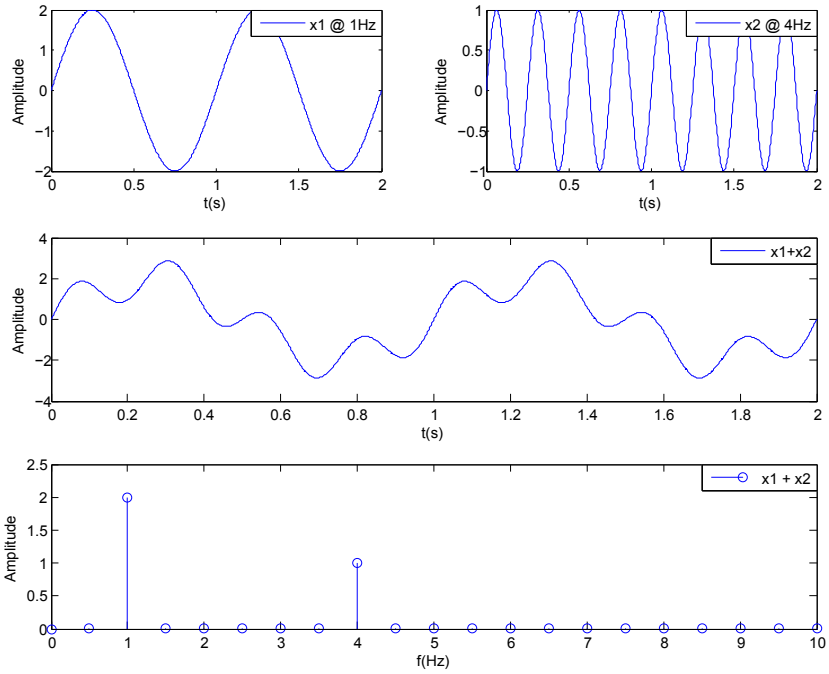


Figura 3.4: Domínio do tempo vs. Domínio da Frequência

3.2.1 Fundamentos teóricos

Do trabalho [34] realizado por Okan K. Esroy podemos retirar as informações descritas ao longo deste tópico.

O processo descrito em 3.2 pode ser realizado utilizando amostragem real ou complexa. As expressões 3.3 e 3.4 descrevem matematicamente a transformada de fourier real (RFT) e a transformada de fourier complexa (CFT) respectivamente. $x(t)$ é um sinal contínuo no domínio do tempo. $X(f)$ e $X_c(f)$ são o conteúdo espectral do sinal no domínio da frequência.

$$X(f) = 2\int_{-\infty}^{+\infty} x(t)\cdot\cos(2\pi ft+\theta(f))dt, \theta(f) = \begin{cases} 0 & , f \geq 0 \\ \pi/2 & , f < 0 \end{cases} \tag{3.3}$$

$$X_c(f) = \int_{-\infty}^{+\infty} x(t)\cdot e^{-j2\pi ft}dt \tag{3.4}$$

Observando os elementos da equação 3.3, conclui-se que é possível apenas com um integral analisar as componentes co-seno e seno de um sinal variando apenas a fase. Devido aos valores estabelecidos para $\theta(f)$ as frequências positivas correspondem ao co-seno, enquanto que as negativas ao seno. A metade negativa do espectro não tem por isso qualquer significado físico.

Quando se trata de processamento digital de sinal, $x(t)$ é amostrado e consequentemente torna-se $x(n)$, em que n indica o número da amostra. O resultado desta substituição é conhecido como transformada de Fourier discreta no tempo (DTFT). Tal como acontece com a FT, existem duas formas: real e complexa.

Partindo da equação 3.3 chegamos ao seguinte conjunto de expressões para a amostragem real (RDTFT):

$$X(f) = \frac{2}{F_a} \sum_{k=-\infty}^{+\infty} x(k) \cdot \cos(2\pi f T_a k + \theta(f)), \theta(f) = \begin{cases} 0 & , f \geq 0 \\ \pi/2 & , f < 0 \end{cases} \tag{3.5}$$

assim como da transformada inversa (IRDTFT):

$$\begin{aligned} X_1(f) &= \frac{2}{F_a} \sum_{k=-\infty}^{+\infty} x(k) \cdot \cos(2\pi f T_a k) \\ X_0(f) &= \frac{2}{F_a} \sum_{k=-\infty}^{+\infty} x(k) \cdot \sin(2\pi f T_a k) \\ x(n) &= \int_0^{F_a/2} (X_1(f) \cdot \cos(2\pi n T_0 f) + X_0 \cdot \cos(2\pi n T_0 f))\, df \end{aligned} \tag{3.6}$$

Partindo da equação complexa 3.4 obtemos 3.7 para a transformada (CDTFT) e 3.8 para o inverso (ICDTFT).

$$X_c(f) = \frac{1}{F_a} \sum_{k=-\infty}^{+\infty} x(k) \cdot e^{-j2\pi f k T_a} \tag{3.7}$$

$$x_c(n) = \int_{-F_a/2}^{F_a/2} X_c(f) \cdot e^{-j2\pi f n T_a} \tag{3.8}$$

O resultado da transformada é constituído por uma parte real e uma parte imaginária formando um número complexo. Com recurso às equações 3.9 e 3.10 podem ser recuperados os valores de amplitude e fase respectivamente.

$$A_c(f) = |X_c(f)| = \sqrt{Re(X_c(f))^2 + Im(X_c(f))^2} \tag{3.9}$$

$$\varphi_c(f) = arg(X_c(f)) = atan2(Re(X_c(f)), Im(X_c(f))) \tag{3.10}$$

3.2.2 Amostragem real versus complexa

É necessário fazer uma comparação das vantagens e desvantagens de cada um dos métodos de amostragem para que se possa escolher o equipamento de forma consciente. Ambas oferecem vantagens e desvantagens na implementação de um receptor.

Começando pela amostragem real, a sua principal vantagem é a simplicidade do equipamento de amostragem mas por consequência surge o problema da ambiguidade das frequências detectadas devido ao fenómeno de espelho observado na equação 3.3. A utilização de amostragem complexa anula este efeito. No entanto o equipamento torna-se mais complexo pois as componentes real e imaginária têm que ser amostradas em separado e de forma coerente. Em termos de largura de banda amostrada também a amostragem complexa leva vantagem. Sendo capaz de distinguir entre frequências negativas e positivas, podemos observar o dobro da largura como se pode confirmar comparando as equações 3.6 e 3.8. Outra vantagem prática da distinção entre frequências negativas e positivas é o facto de facilmente se verificar se estamos a sintonizar acima ou abaixo da frequência alvo quando temos uma conversão de frequência.

3.2.3 Fast Fourier Transform (FFT)

De todos os métodos utilizados para calcular a DFT, a FFT é de longe o mais utilizado: pode beneficiar de algoritmos que diminuem substancialmente o cálculo o que é essencial para funcionar em tempo real. Se tentarmos calcular a DFT usando directamente a equação 3.7 iremos necessitar de N^2 operações. Ao recorrermos à FFT vemos esse número reduzido para $N\log(N)$ em qualquer dos algoritmos conhecidos.

O método radix-2 Cooley Tukey é um dos mais rápidos utilizando apenas $(N/2)\log_2(N)$ multiplicações complexas e $N\log_2(N)$ adições. O número de amostras para a utilização deste algoritmo tem que respeitar a forma 2^n. Isto acontece porque o conjunto de valores da DFT é subdividido em dois. Cada uma das metades é novamente divida em dois. O processo repete-se $n-1$ vezes. Estas divisões permitem o cálculo de DFTs mais pequenas e por isso rápidas. Cada valor calculado corresponde a um ponto da DFT geral também conhecido como *bin*. Cada *bin* representa a energia contida dentro da largura de banda de resolução. A resolução da DFT depende por isso do número de subdivisões feitas.

O erro associado a este método de compressão logarítmica é dado por $\varepsilon\log(N)$, em que ε é a precisão relativa da vírgula flutuante na máquina em que está a ser executado o cálculo.

Um dos problemas da utilização da FFT é o alargamento do espectro. Não sendo possível ter uma distância infinitesimal entre pontos a energia é distribuída entre *bins* adjacentes.

Existem funções que ajudam na correcção deste efeito, são conhecidas como funções de *windowing*. O gráfico da figura 3.5 apresenta uma forma típica de uma função *window* assim como o resultado da sua aplicação.

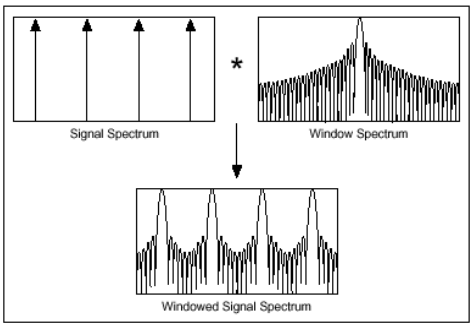


Figura 3.5: Aplicação da função *window*

A figura 3.6 apresenta os principais parâmetros de uma função de *window*. Existem diferentes funções *window* e devem ser escolhidas com base nestes parâmetros. Algumas estratégias para a escolha são descritas em [38].

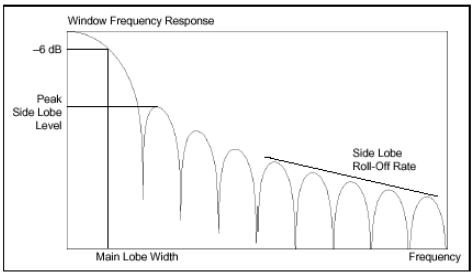


Figura 3.6: Parâmetros da função *window*

3.3 Algoritmos

O sistema implementado recorre a análise espectral para estimar a frequência e fase relativa entre o sinal *copolar* e *crosspolar* assim como as estatísticas associadas. Por este motivo é necessário efectuar um estudo prévio sobre os algoritmos a utilizar.

A FFT é actualmente a ferramenta mais versátil para cálculo no domínio da frequência, tornando-a a escolha ideal para servir de base a todos os algoritmos estudados.

3.3.1 Estimativa de potência

A estimativa da potência não é um processo imediato como seria de esperar à primeira vista.

O primeiro problema surge com o espalhamento de potência ao longo do espectro devido ao ruído de fase do *beacon* e ao ruído branco aditivo. A solução para este problema passa pela escolha das riscas de interesse e consequente somatório da potência por elas contida. No entanto existe uma limitação que se prende com a resolução da FFT. A precisão com que são definidos os limites da banda que se pretende utilizar depende da largura de cada uma das *bins* que a compõem.

Caso o ruído seja muito inferior ao sinal em estudo, isto não será um problema relevante na medida em que o total de potência adicionado devido à detecção de ruído, não introduz um erro considerável no valor final de potência do sinal. À medida que a CNR se degrada o problema da resolução da FFT acentua-se, tornando-se necessário dispender um esforço adicional na escolha do número de pontos da FFT.

Neste âmbito, foram realizadas simulações em MATLAB para avaliar a influência da resolução de frequência na estimativa da potência total do sinal. Os resultados do desempenho são apresentados nos gráficos da figura 3.7. Estes mostram a variância da potência estimada utilizando a equação 3.11.

$$P_{sinal} = \sqrt{\sum_{i=1}^{n_{bins}} (co_i \cdot co_i^*)} \tag{3.11}$$

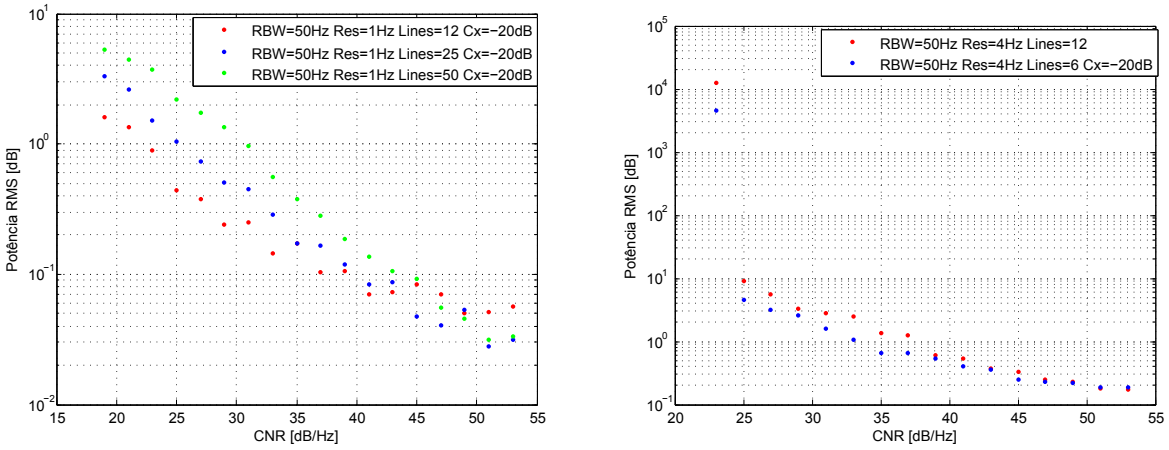
Pode-se observar que a utilização de uma resolução mais fina conduz a melhores resultados no que diz respeito à estimativa da potência. No entanto, efectuando a média dos resultados obtidos para a resolução de 4Hz constata-se que a sua variância se aproxima da registada para 1Hz. A escolha do número de riscas que definem a largura espectral que contém a potência do sinal é um ponto crucial. É também possível observar que, por um lado, caso a CNR seja elevada a utilização de um maior número de riscas diminui o erro. Por outro lado, utilizar menos riscas torna-se vantajoso no momento em que a CNR se deteriora.

Um exemplo prático consiste no cálculo da potência contida numa largura de 15Hz, em que as escolhas de resolução possíveis são 2Hz e 10Hz por risca.

Por um lado, ao utilizar a resolução de 2Hz, a menor largura de banda que contém totalmente os 15Hz é composta por 8 riscas, o equivalente a 16Hz. O que corresponde a incluir 1Hz de ruído não pertencente à banda para a estimativa de potência, resultando numa estimativa por excesso. Por outro lado, poderíamos utilizar apenas 7 *bins* (14Hz), em que a potência de 1Hz do espectro do sinal não seria contabilizada, efectuado assim uma estimativa por defeito.

Analisando a hipótese de uma resolução de 10Hz, conclui-se que resulta em valores menos precisos. Isto porque escolhendo 2 *bins* o equivalente a 20Hz, estimativa por excesso, estaríamos a incluir 5Hz de ruído lateral no somatório. Aplicando uma estimativa por defeito utilizando uma *bin*, 10Hz de largura, implica deixar a potência do 5Hz de sinal fora do somatório.

Em suma, o valor ideal de resolução vai depender em grande parte da largura de banda do sinal a analisar, podendo ser negligenciada para casos em que a CNR seja suficientemente elevada. Também se pode concluir que caso seja necessário a escolha cuidada da resolução da FFT, esta é deixada a cargo



(a) 1Hz (b) 4Hz

Figura 3.7: Variância da estimativa de potência para diferentes resoluções

de quem desenvolve o sistema, sendo sempre preferível a mais fina possível para obter resultados mais confiáveis. Em relação ao número de riscas, o número deve ter em conta a CNR do sinal a analisar.

3.3.2 Estimativa de frequência

A estimativa da frequência é um ponto fulcral na realização de um detector de sinal pois poderá no futuro vir a ser implementado um seguimento de frequência.

Na figura 3.8 é apresentado a azul o espectro de um sinal, e a vermelho a sua FFT com 10Hz de resolução. Como se pode observar, o valor exacto da frequência de pico é perdido durante o cálculo da FFT. O facto de uma risca representar a potência contida numa determinada gama de frequências, impede que se retire directamente a frequência na qual o sinal se encontra centrado olhando apenas para a *bin* de maior potência.

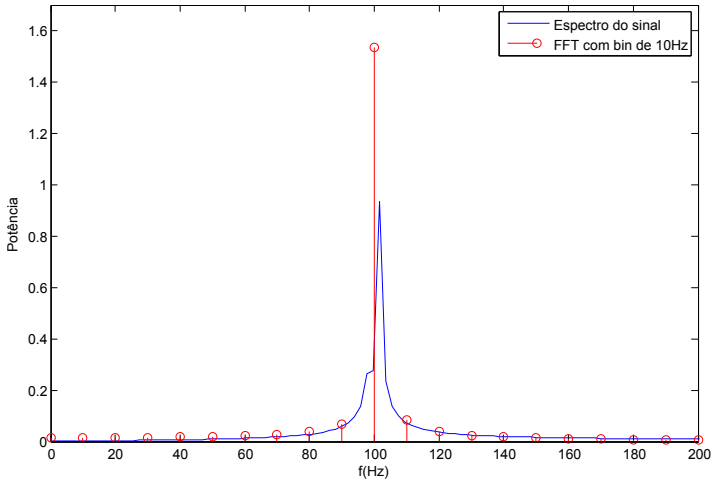


Figura 3.8: Espectro Real vs. FFT *bins*

Este problema pode ser atenuado de duas maneiras: aumentando o número de pontos, reduzindo a margem de erro entre a frequência da *bin* e o valor real; através do cálculo de uma média pesada da frequência da *bin* de maior potência juntamente com algumas adjacentes. A segunda opção pode ser utilizada em conjunto com a primeira para melhorar ainda mais a precisão final.

Olhando para a primeira solução encontramos dois problemas óbvios. Por um lado temos o aumento dos requisitos computacionais para o cálculo de uma FFT de maior dimensão. Isto não causará transtornos se o sistema em questão tiver recursos livres suficientes para carga de processamento adicional. Por outro lado, uma FFT maior requer um maior número de amostras, levando a um período de espera entre estimativas mais longo. Dependendo da finalidade, esta poderá estar fora de questão na medida em que diminui a velocidade de resposta do sistema a variações de sinal, reduzindo assim a sua agilidade.

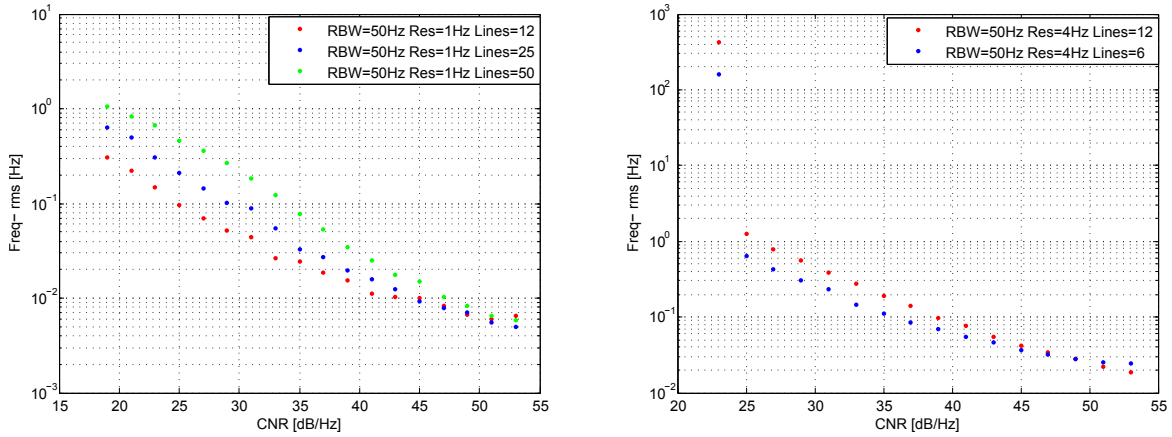
A segunda opção, embora não esteja livre de problemas, em alguns casos apresenta-se como a melhor solução. De maneira geral, o número de operações para o cálculo de uma média pesada, usando a equação 3.12, $(N+1)+2\cdot(N-1)$, é inferior ao necessário para o cálculo de uma FFT de maior resolução, $N\log(N)$.

$$f_c = \frac{\sum_{i=1}^{n_{bins}} (bin_i \cdot bin_i^* \cdot f_{bin_i})}{\sum_{i=1}^{n_{bins}} (bin_i \cdot bin_i^*)} \tag{3.12}$$

O maior problema da implementação de uma média pesada para recuperar o valor da frequência central é a escolha do número de riscas a incluir. Isto deve-se à influência da potência do ruído em situações em que a CNR é demasiado próxima de $10\log(B)$, sendo B é a largura de banda em análise.

Na imagem 3.9 são apresentados os resultados dos desvios obtidos nas simulações efectuadas, onde se comparam as duas opções propostas. Em cada um são confrontados os valores utilizando um número diferente de riscas para o cálculo da média pesada. No gráfico 3.9(a) é utilizada uma resolução de 1Hz enquanto que no gráfico 3.9(b) é utilizado 4Hz.

É por isso possível tirar conclusões acerca da influência do número de riscas a utilizar assim como da resolução das mesmas.



(a) 1Hz (b) 4Hz

Figura 3.9: Variância da estimativa de frequência para diferentes resoluções

Uma das conclusões a retirar é que um número mais reduzido de riscas para uma mesma resolução espectral, melhora a precisão da estimativa em situações de baixa CNR. Isto deve-se a uma menor influência do ruído na amplitude das riscas. Escolhendo menos riscas em torno da central garante-se que a qualidade espectral é superior na medida em que a contaminação pelo ruído é menor. À medida

que as riscas se afastam do ponto central a sua potência é cada vez mais determinada pelo ruído e não pelo sinal.

A segunda conclusão é que uma melhor resolução conduz a uma maior precisão como era esperado. Tal como foi explicado, o facto da gama de frequência representada pela *bin* ser mais estreita permite que a frequência seja descriminada com maior detalhe.

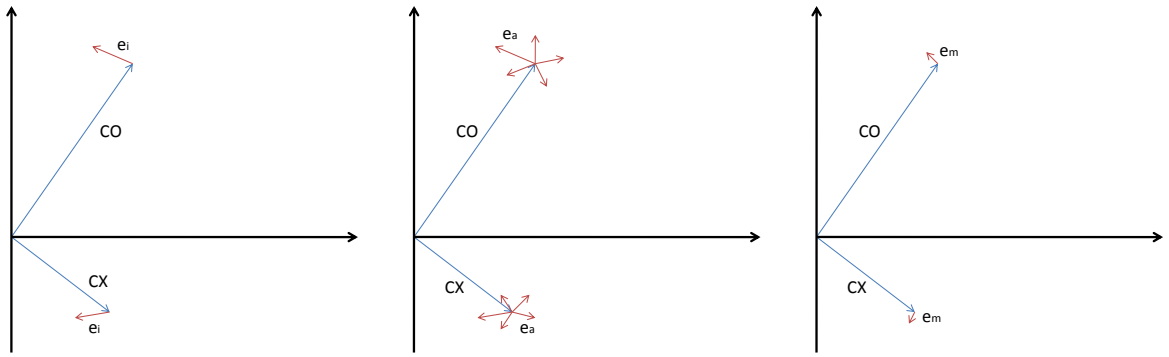
Concluindo, o critério utilizado para fazer o balanceamento destes parâmetros fica novamente a cargo do responsável pelo desenvolvimento do sistema. A escolha deve ser feita tendo em conta a velocidade de resposta do sistema a variações do sinal, o nível de precisão requerido pela aplicação, os recursos computacionais disponíveis e o ruído de fase do *beacon*.

3.3.3 Estimativa de fase relativa entre dois sinais correlacionados

O cálculo da fase relativa entre dois sinais correlacionados é uma tarefa simples. No entanto isto não é verdade quando os vectores são afectados por ruído. A figura 3.10(a) ilustra dois vectores sinal, CO e CX, e o ruído adicionado, e_i. Após soma do ruído verifica-se que a fase de ambos os vectores de sinal é alterada, assim como a sua amplitude. No limite o vector erro estará desfasado $\pi/2$ em relação aos sinais, o que se traduz na máxima rotação. Como se pode deduzir, esta situação conduz aos piores resultados possíveis. No melhor dos casos o ruído está em fase ou em oposição de fase, alterando apenas a amplitude e mantendo constante a fase.

Uma das maneiras possíveis de contrariar este efeito, tem como base a realização de médias das amostras. Isto é possível porque por um lado, considerando que o ruído branco é o dominante na natureza e no sistema, utiliza-se umas das suas características em nosso favor: a sua média tende para um valor nulo. Quanto mais amostras forem utilizadas no cálculo mais próximo de zero será o resultado. Por outro lado, tanto o sinal *copolar* como o *crosspolar* variam lentamente e podem ser considerados constantes para períodos de amostragem de alguns segundos.

As figuras 6.2(a) e 6.2(b) apresentam as duas fases deste processo, primeiro a recolha de várias amostras e seguidamente a média das mesmas. A comparação entre ambas reflecte de forma clara o melhoramento imposto por esta técnica.



(a) Instantânea (b) Amostrada (c) Média

Figura 3.10: Vectores de sinal e ruído para diferentes medições

Em primeira análise podemos concluir que o sistema será mais lento a devolver o valor da fase relativa, implicando sempre um período de amostragem seguido da computação da média. Esta desvantagem é largamente compensada pela maior imunidade ao ruído adicionado ao sinal.

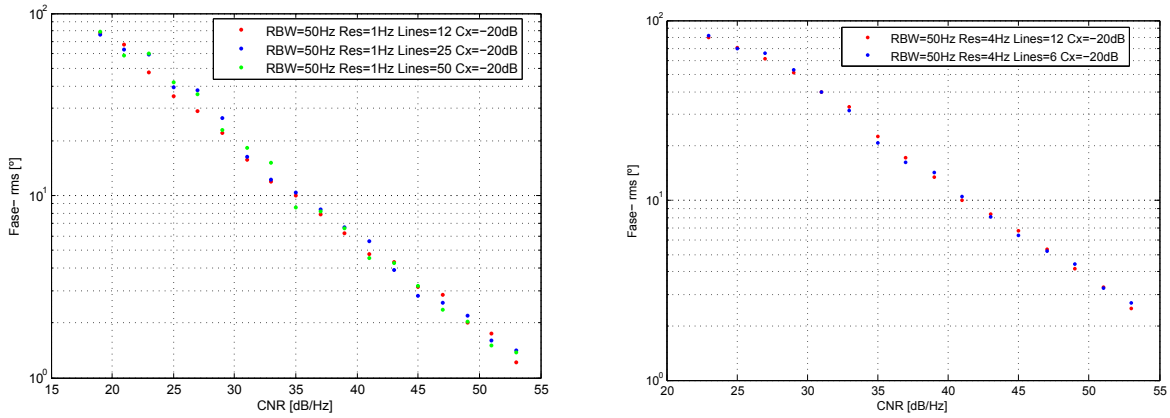
Na prática, isto é implementado através do cálculo de duas FFTs, para os sinais CO e CX. Como estas não se baseiam apenas numa amostra, mas num conjunto adquirido ao longo do seu período, o ruído será atenuado pelo motivo explicado anteriormente.

O passo seguinte é determinar a *bin* correspondente ao máximo de potência do sinal de referência, neste caso o *copolar*. Conhecendo este ponto é feita a soma dos valores complexos da *bin* central e de algumas adjacentes a esta. Esta soma apresenta de forma intrínseca um peso associado a cada uma das *bins*. Isto é, quanto maior for o módulo maior será a influência na fase do vector resultante da soma. Ao mesmo tempo, as componentes de ruído são novamente diminuídas devido ao efeito observado nos esboços da figura 3.10. Como o sinal $crosspolar$ está centrado na mesma frequência que o $copolar$, são utilizados os mesmos índices para o seu somatório.

Em seguida, é recuperada a fase relativa recorrendo à função $\arctan 2(y, x)$ que determina a fase de um vector fazendo distinção entre quadrantes. A equação devolve o valor do ângulo entre o *copolar* e o *crosspolar* é a seguinte:

$$\theta = \arctan 2 \left(imag \left(\sum_{i=1}^{n_{bins}} (cx_i \cdot co_i^*) \right), real \left(\sum_{i=1}^{n_{bins}} (cx_i \cdot co_i^*) \right) \right) \tag{3.13}$$

Para determinar a influência que a resolução da FFT e o número de *bins* adjacentes seleccionadas tem na estimativa da fase, foram realizadas algumas simulações em MATLAB. Os gráficos da figura 3.11 resumem os resultados obtidos, apresentando o desvio médio ao quadrado da fase estimada em relação à esperada.



(a) 1Hz (b) 4Hz

Figura 3.11: Variância da estimativa de fase relativa para diferentes resoluções

Podemos concluir através dos gráficos que o número de riscas adjacentes escolhidas não tem qualquer influência na estimativa. No entanto a utilização de um período mais longo na FFT conduz a resultados melhores. Estes dados corroboram as previsões de que um maior número de amostras conduz a valores mais confiáveis.

3.3.4 Estimativa de amplitude de um sinal *crosspolar*

A estimativa do sinal *crosspolar* não pode ser realizada directamente através da soma da potência das suas *bins* pois é necessário garantir a sua correlação com o *copolar*.

O algoritmo implementado utiliza como factores de correlação a gama de frequências do espectro do sinal *co* e as suas respectivas amplitudes. Adicionalmente é necessário conhecer *a priori* o desfasamento relativo calculado usando o método descrito em 3.3.2.

A equação 3.14 devolve o valor de amplitude correlacionando ambos os sinais.

$$A_{cx} = \frac{\left| \sum_{i=1}^{n_{bins}} (cx_i \cdot e^{j(3\pi/2-\theta)} \cdot co_i^*) \right|}{\sqrt{\sum_{i=1}^{n_{bins}} (co_i \cdot co_i^*)}} \tag{3.14}$$

Existem três aspectos que devem ser salientados na equação usada, nomeadamentem a rotação de fase, a multiplicação do sinal *cx* e a normalização.

Em primeiro lugar, a rotação de fase apresentada permite alinhar os vectores $\overrightarrow{co}$ e $\overrightarrow{cx}$ deixando-os em fase ou em oposição de fase. Deste modo é mais simples estabelecer uma relação de amplitude entre ambos.

Em segundo lugar é feita a multiplicação pelo conjugado do *copolar*. É nesta multiplicação que se estabelece a correlação entre ambos. Facilmente pode-se concluir que os valores que irão ter maior peso no somatório serão os que forem multiplicados pelos conjugados de maior módulo. Assim, é dada mais importância às *bins* do *crosspolar* que estiverem alinhadas em frequência com o sinal principal, diminuindo o erro imposto pela potência de ruído em *bins* adjacentes.

Por último a divisão permite recuperar o valor normalizado da amplitude.

3.3.5 Estimativa da densidade espectral da relação sinal ruído e CNR

Um dos valores importantes de registar que se pode estabelecer é a relação sinal ruído do sinal ao longo do tempo, que é um indicador da qualidade de sinal que tem em consideração a potência total do sinal. Esta relação é dada pela expressão 3.15.

$$CNR = 10 \log \left(\frac{P_{sinal}}{NSD} \right) \tag{3.15}$$

Após observar a equação 3.15 conclui-se que é necessário determinar o NSD em primeiro lugar. O NSD é o valor de potência de ruído por Hertz. Este é relativamente simples de obter utilizando uma FFT. Para tal basta saber a largura do espectro do sinal a observar e utilizar uma FFT com uma banda de amostragem maior. Enquanto que um determinado número de riscas corresponde à potência do sinal espalhada ao longo do espectro, as riscas restantes contém maioritariamente ruído. Assim, ao somar a potência das *bins* de ruído e dividir pela largura de banda que elas representam obtemos o NSD.

A potência do sinal é calculada através de um processo semelhante utilizando as riscas referentes à frequência central das bandas laterais como descrito no tópico 3.3.1.

Tal como os algoritmos descritos anteriormente, também aqui o número de riscas influencia a estimativa do NSD. Como é de esperar, sendo o ruído um processo aleatório, a utilização de uma banda maior produz uma média mais próxima do valor real de NSD.

A escolha do número de riscas para estimar a potência de sinal já foi discutida no tópico 3.3.1, e por isso não serão feitos quaisquer comentários adicionais.

3.3.6 Cálculo recursivo da variância

A variância é dada pelo desvio padrão ao quadrado (σ^2) e permite quantificar a estabilidade de um sinal a longo prazo.

O desvio padrão é dado pela diferença entre entre o valor da estimativa actual em relação ao valor médio de todas as amostras.

$$\sigma = \sqrt{\sum (x - \overline{x})^2} \tag{3.16}$$

A média do sinal pode ser calculada simplesmente mantendo dois registos que permitam saber o número de estimativas registadas e a sua soma ao longo do tempo.

Para que seja possível obter o valor da variância à medida que novas estimativas são registadas é utilizada a seguinte expressão, em que n representa o número da amostra.

$$\sigma_n^2 = \sigma_{n-1}^2 \cdot \left(\frac{n-1}{n}\right) + \sigma_{amostra}^2 \cdot \left(\frac{1}{n}\right) \tag{3.17}$$

Deste modo é atribuído um peso de $(n-1)/n$ à variância das amostras passadas, e um peso de $1/n$ à variância calculada para a última amostra.

Inicialmente o valor de σ_0^2 é zero pois ainda só existe uma amostra feita. Para cada amostra seguinte e consequente estimativa associada é actualizado o valor da soma e de quantidade de amostras realizadas até ao presente momento. Com estes dois valores actualizados podemos calcular σ_n^2 e usar a equação 3.17 para obter a variância do sinal.

Este é um método simples de calcular a variância de um sinal em tempo-real. A limitação deste método prende-se com o valor máximo que a soma pode atingir tendo em conta o tamanho da variável que a acomoda. No entanto, estamos apenas interessados em saber a variância para períodos relativamente curtos de tempo, e por este motivo a soma nunca resulta em casos de *overflow*.

3.4 Digital down conversion (DDC)

A translação da banda de um sinal para outra IF é um processo essencial para o processamento digital de sinal. Tal como foi descrito em 2, um sinal de satélite nunca é transmitido em banda base. Por esse motivo é necessário fazer essa conversão para facilitar o seu processamento quer em termos da frequência de amostragem utilizada assim como na implementação de filtros.

A figura 3.12 representa o equivalente digital de um clássico receptor super heteródino, o seu congénere analógico.

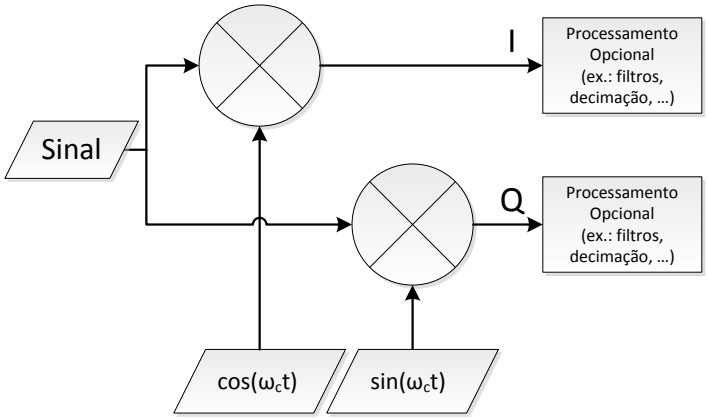


Figura 3.12: Diagrama de um DDC

É feita uma amostragem complexa. O sinal é multiplicado por $\cos(\omega_c t)$ para obter a parte real e por $\sin(\omega_c t)$ para obter a parte imaginária que se encontra desfasada de $90°$. Aplicando a transformada

de Fourier a um sinal que foi multiplicado por uma sinusóide de frequência ω_c, verifica-se o espectro do sinal sofre uma translação igual a $-\omega_c$ no domínio da frequência. Esta propriedade é usada a nosso favor para implementar digitalmente uma conversão de uma IF (no nosso caso 10.7MHz) para uma frequência áudio na qual, após decimação adequada, será possível realizar FFT com uma resolução espectral suficientemente fina mas de tamanho aceitável.

A geração do sinal seno e co-seno é, em geral, feita com recurso a um oscilador de controlo numérico (NCO). Este tem acesso a uma ROM (ou algoritmo CORDIC) onde se encontram os valores de amplitude para as diferentes fases do um sinal sinusoidal. O NCO está ligado a um oscilador local e possuí um contador. É então programado externamente para que dê M passos por cada ciclo de relógio. A resolução máxima depende do tamanho da tabela gravada na ROM assim como da velocidade do relógio local como se mostra na equação 3.18, em que f_o é a frequência gerada, f_c é a frequência do oscilador local e 2^n é o tamanho da tabela armazenada na ROM.

$$f_o = \frac{M \cdot f_c}{2^n} \tag{3.18}$$

Mais informações sobre o funcionamento deste componente são apresentadas em [33].

3.5 Filtros

Tal como foi mencionado no tópico 3.1.1, é necessário a implementação de filtros que, por um lado reduzam o efeito de componentes cuja frequência é superior a $f_a/2$ e por outro lado eliminam as imagens que surgem após a amostragem do sinal.

Existem dois tipos principais de filtros digitais: filtro de resposta finita (FIR) e filtro de resposta infinita (IIR). O filtro em que nos vamos focar é o FIR. Um exemplo básico da sua implementação é feito na figura 3.13 que se traduz pela função de transferência 3.19.

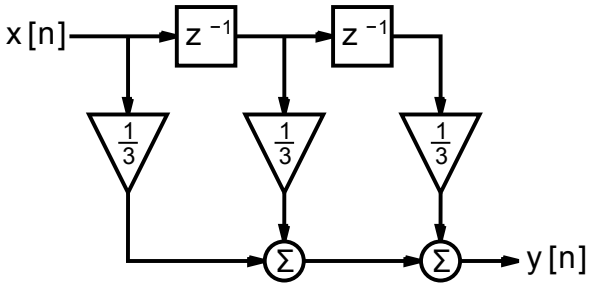
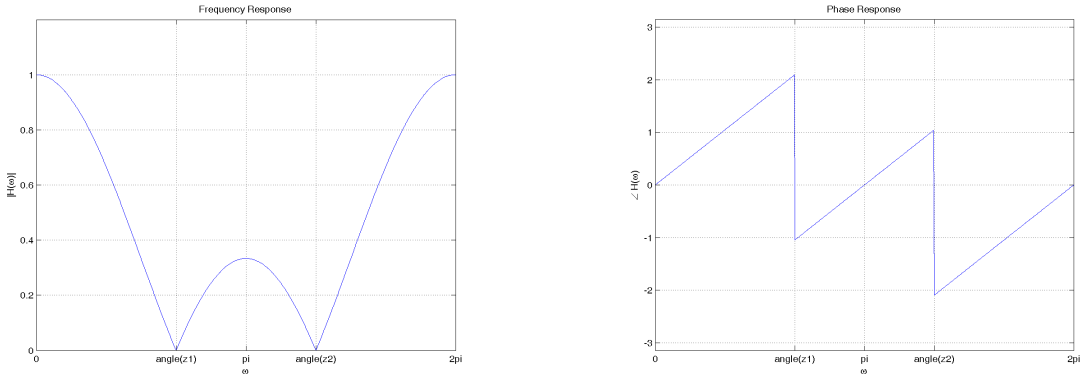


Figura 3.13: Diagrama de um filtro FIR

$$H(f) = \frac{1}{3}\left(\frac{z^2 + z + 1}{z^2}\right) \tag{3.19}$$

Observando a imagem vemos que se trata de um filtro simples em malha aberta. Por não ter realimentação os erros de arredondamento não são acumuláveis. O mesmo não acontece com os filtros IIR devido à realimentação, tornando-se potencialmente instáveis. O filtro consiste numa série de atrasos (z^{-1}) aplicados às amostras na entrada. Cada um dos atrasos é multiplicado por um coeficiente que atenua as amostras mais antigas, conferindo-lhe uma característica passa-baixo como ilustrado na figura 3.14(a). Por fim todas as multiplicações são somadas para dar origem a um dos valores de saída da função de transferência do filtro $H(f)$. As frequências superiores a π tratam-se de imagens das frequências abaixo de π.



(a) Resposta da amplitude

(b) Resposta da fase

Figura 3.14: Resposta do filtro da figura 3.13

Capítulo 4

Software Defined Radio

4.1 Conceitos básicos

O principal objectivo do *software defined radio* é substituir, dentro do fisicamente possível, todos os processos analógicos desempenhados por um equipamento de radio comum.

O funcionamento de qualquer aparelho electrónico pode ser descrito por uma ou mais equações matemáticas. Na realidade cada elemento dessas equações é traduzido para o mundo físico por um componente cuja resposta a um determinado estimulo se assemelhe à função que representa. Por outro lado sabendo as equações que definem um determinado sistema é teoricamente possível simulá-lo no mundo digital através de cálculos matemáticos. Empiricamente sabe-se que isto não é possível para alguns tipos de sistema, que devido à sua complexidade tornam as simulações numéricas impracticáveis.

Actualmente grandes progressos têm sido feitos na área do *hardware* de processamento digital. Por um lado os computadores dos dias de hoje são dezenas de vezes mais rápidos que os utilizados há alguns anos atrás. Por outro lado o aparecimento de *field programmable gate arrays* (FPGA) veio em muito contribuir para o *hardware* digital ser utilizado cada vez mais em RF.

Uma FPGA, no fundo, trata-se de um processador re-programável. Isto permite retirar os benefícios da velocidade de uma implementação em *hardware* e da flexibilidade do *software*. Em geral este componente é programado para executar tarefas cujos cálculos são demasiado intensos, mas repetitivos, para serem realizados por um processador genérico. Desta forma é possível, por exemplo, implementar filtros digitais de grande complexidade a baixo custo sem afectar o desempenho da máquina que fará o tratamento *útil* do sinal.

Outra vertente que têm vindo a ser desenvolvida é a conversão de unidades de processamento gráfico (GPU) para processamento genérico (GPGPU). Dois exemplos deste esforço são apresentados pela nVidia [20] e pela AMD [3], dois dos maiores fabricantes de placas gráficas a nível mundial. Embora seja conhecido o facto de que este tipo de processador ter uma maior capacidade de processamento em vírgula flutuante que um genérico, nenhum foi concebido com as instruções de processamento geral em mente, sendo indicados maioritariamente para processamento matricial como é o caso de uma imagem.

Tudo o que foi dito nos últimos parágrafos é um indicativo de que cada vez mais é possível a simulação em tempo real das equações complexas que definem os sistemas analógicos que queremos substituir, e que a tendência para a evolução deste tipo de processadores é bastante elevada.

Em teoria um sistema SDR ideal teria o esquema apresentado na figura 4.1. Embora teoricamente ele seja possível, tal não acontece na vida real.

O primeiro erro cometido é admitir que o ADC é capaz de acompanhar a variação de frequência do sinal recebido pela antena. Por exemplo, no nosso caso a frequência recebida é na ordem dos

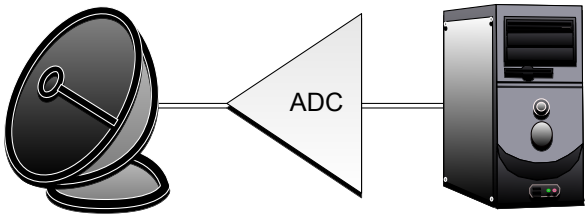


Figura 4.1: Diagrama de um SDR ideal

40GHz. Não existe qualquer tipo de tecnologia digital que trabalhe a frequências tão elevadas. Por esse motivo é necessário introduzir um *frontend* RF entre a antena e o ADC. Este *frontend* tem como função principal reduzir a frequência do sinal recebido para uma mais baixa, possível de amostrar digitalmente e ler através de uma ligação ao computador. Duas das funções secundárias mais comuns são a amplificação e filtragem do sinal de forma a poder ser detectado e convertido pelo ADC.

O segundo erro é admitir que se pode fazer uma leitura directa de um ADC, especialmente quando as frequências de amostragem são elevadas. Admitindo que se está a analisar um canal de TV cuja largura ronda os 8MHz o ADC terá que funcionar pelo menos a 16MHz. Considerando que as amostras são feitas com 10 bits de resolução (relativamente baixa), isto corresponde a um ritmo de dados de cerca de 20MBytes/s no mínimo. Partindo do princípio que uma melhor resolução é necessária para diminuir os erros de arredondamento, a utilização de uma frequência de amostragem maior para canais de banda mais larga assim como a existência de múltiplos canais, chega-se à conclusão de que uma grande quantidade de informação é gerada a cada segundo que passa. De modo a garantir uma transferência contínua e sem falhas entre o ADC e o computador que irá analisar os dados é necessário fazer a decimação e filtragem do sinal digitalizado. Esta função é ideal para implementar com uma FPGA. Recorrendo a uma FPGA podem ser aplicados filtros digitais e decimação de forma transparente ao destinatário final, o que leva a que o desempenho dos cálculos que emulam o *hardware* analógico não seja afectada.

Depois destas observações, conclui-se que o sistema básico para implementar um SDR tem o aspecto da figura 4.2.

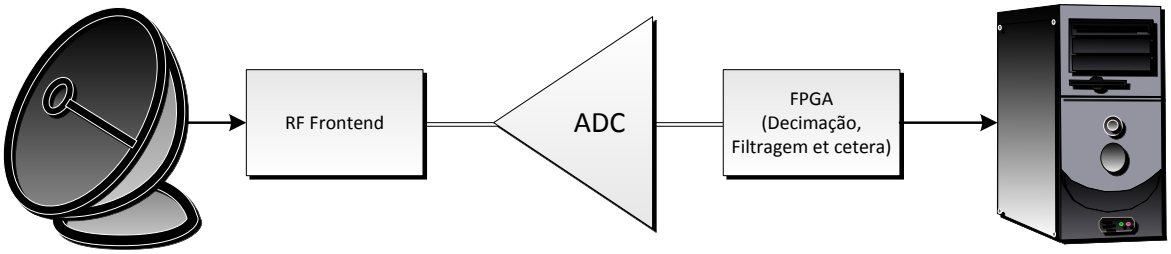


Figura 4.2: Diagrama de um SDR real

4.2 Vantagens de um sistema SDR

Depois de explicado o funcionamento de um sistema desta natureza ficam claras as suas vantagens. Ao substituirmos os componentes analógicos por algoritmos em *software* vamos transformar problemas de natureza física para matemática pura. Um rádio cujo funcionamento é baseado unicamente em *hardware* torna o seu desenvolvimento um processo custoso e demorado.

Em parte é custoso não apenas pelo maior grau de trabalho envolvido em criar por exemplo um filtro ou um misturador, mas também pela maior dificuldade em efectuar os testes do mesmo. No caso

do desempenho do protótipo da peça não respeitar os parâmetros de funcionamento que se esperavam, mais horas terão que ser dispendidas para descobrir o porquê da sua falha. E, pelo facto de se tratar de um elemento físico a falha poderá não estar no esquema eléctrico ou nos cálculos, mas sim num defeito de fabrico ou qualquer outro factor ligado à sua produção. Por este motivo os testes de *hardware* são mais complicados e por isso requerem mais tempo. Como tempo é dinheiro, quanto mais demorada for a fase de testes mais tempo levará o sistema a ser colocado no mercado de modo a recuperar o investimento feito em investigação e desenvolvimento (I&D). Para além do tempo extra necessário para obter uma versão final do produto temos também os elevados custos de prototipagem. Apenas com a produção em massa é possível reduzir os custos dos sistemas que se estão a desenvolver. Enquanto não se tornar um produto acabado cada falha de desenvolvimento implica a encomenda de um novo e redesenhado protótipo a preços bastante elevados.

Ao utilizar um rádio cujos componentes são emulados através de algoritmos temos a vantagem de os erros de desenvolvimento serem mais fáceis de reparar e testar. Se qualquer erro for detectado durante os testes é possível, através de *debug*, saber a zona do código onde ele ocorreu e alterar rapidamente para efectuar um novo teste. Isto simplesmente não é possível de fazer com circuitos integrados (IC).

Para além do ponto de vista económico temos o tecnológico. Esta área também apresenta grandes vantagens em relação a um circuito analógico. Novamente, pelo facto de ser um algoritmo que emula um componente, um simples trocar de parâmetros faz com que se torne em algo com um comportamento bastante diferente. Exemplificando com o caso de um transístor, o ganho β é estabelecido pelo fabricante e não pode ser alterado pelo engenheiro ou técnico que o utilizará. No entanto em software, após definir as equações que replicam a reacção do transístor a um estímulo na entrada, facilmente podemos indicar que aquele transístor deixou de ter um β de 300 mas sim de 150. O que fisicamente equivale a trocar um componente do circuito. Se eventualmente fosse vantajoso fazer uma alteração desta natureza num sistema, era impossível de o conseguir em tempo real num sistema analógico. Seria possível fazer uma implementação semelhante através de um sistema que comutasse entre transístores. No entanto é impossível incluir um grupo de transístores com uma diversidade de parâmetros tão elevado como acontece numa implementação em *software*. Como agravante, a complexidade de tal sistema em *hardware* seria um factor de peso nos custos de desenvolvimento.

Se virmos este problema do ponto de vista do *software*, não seria um grande desafio a alteração deste parâmetro em tempo real, sendo possível variações mais precisas e flexíveis do que utilizando elementos físicos. Este é um exemplo simples da flexibilidade proporcionada por um sistema SDR. Qualquer parâmetro do sistema pode ser alterado em tempo real, se virmos que tal é necessário.

Outro ponto positivo a apontar aos sistemas de SDR é a facilidade de *upgrade*. O nosso algoritmo poderá ser tão complexo e preciso quanto a velocidade da máquina que o executa permite. No entanto, todos os anos os processadores ficam mais rápidos e os seus preços descem rapidamente, o que não acontece com equipamento analógico de topo que mantém o seu preço constante por períodos de vida mais longos.

A desvantagem global dos sistemas SDR é o facto de estarem limitados a frequências baixas quando se fala de transmissão rádio, na casa das dezenas de MHz. Esta limitação é imposta principalmente pela velocidade do ADC na entrada. Embora a tendência seja para termos frequências de amostragem nos ADC cada vez maiores, ainda se encontram muito longe do que um *frontend* analógico é capaz de fazer. Porém este mesmo *frontend* vai diminuindo a sua complexidade à medida que é possível a amostragem de IFs a frequências cada vez mais elevadas.

4.3 *Hardware SDR*

Ao longo dos anos várias plataformas de *hardware* têm vindo a ser desenvolvidas de modo a facilitar a entrada no mundo do rádio por *software*. Em [31] é apresentada uma implementação, em meio académico, de um sistema digital para processamento de sinal rádio. No entanto várias opções estão disponíveis no mercado apresentadas por diversos fabricantes. Entre esses equipamentos prontos para utilização encontram-se o SDR-IQ [14], Perseus [17], o QS1R VERB [24], o ADT-200A [1], o USRP1 e 2 [8] e muitos mais.

O USRP foi o sistema que se distinguiu de todos os analisados.

4.3.1 Universal Software Radio Peripheral (USRP)

Em suma, é um conjunto completo de recepção e transmissão com ADCs, DACs, FPGA, comunicação USB2.0 e placas de expansão desenvolvido por Matt Ettus na Ettus Research [8]. O *hardware* é aberto, isto é, estão disponíveis os esquemáticos para o replicar, alterar ou criar placas de expansão personalizadas. Uma descrição mais detalhada pode ser encontrada no documento [37] escrito por Firas Abbas Hamza. Possui quatro entradas analógicas e duas saídas analógicas. quatro encaixes para expansão, dois deles dedicados apenas a recepção e dois apenas para transmissão. Incorpora também uma FPGA para DDC/DUC, filtragem e decimação/interpolação dos sinais de entrada/saída. E ainda, um integrado para transferência de dados usando comunicação por USB2.0.

4.3.1.1 Placa mãe

A figura 4.3 apresenta uma imagem da placa mãe que alberga todos os componentes e encaixes de expansão.

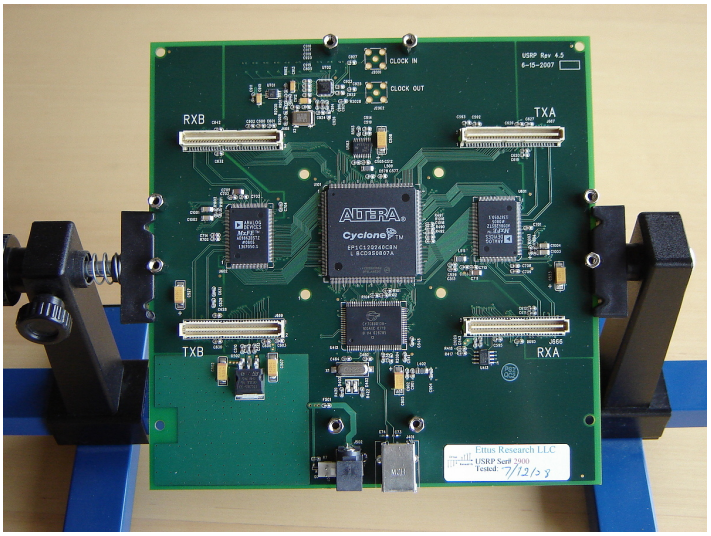


Figura 4.3: USRP - placa mãe

Analisando mais atentamente encontramos dois integrados AD9862 da Analog Devices. Estes *chips* incorporam dois ADCs e dois DACs. Cada conjunto RX/TX partilha um dos integrados AD9862, conferindo dois ADCs e dois DACs a cada ramo.

O ADC é capaz de recolher 64M amostras por segundo a 12 *bits* de resolução, sendo em teoria capaz de analisar uma banda de 32MHz. Pode funcionar como passa-banda para frequências até 200MHz, e caso vários dBs de gama dinâmica sejam permitidos é possível utilizar IFs na ordem dos 500MHz. É necessário ter atenção ao facto de que ao utilizar frequências mais elevadas que 32MHz o

surgimento de frequências imagem é inevitável. A excursão de sinal permitida na entrada é de 2Vpp utilizando uma carga de 50Ω, o que equivale a uma potência máxima de 10dBm. Ligado aos ADCs existe um amplificador de ganho programável (PGA). O seu valor varia entre 0 e 20dB, o que faz com que o sinal máximo seja de 0.2Vpp para o ganho máximo. O PGA é controlado por software.

Os DACs funcionam a 128MHz podendo por isso transmitir até 64MHz. A potência do sinal gerado tem um máximo de 10dBm. Também os DACs estão conectados ao PGA.

Em termos de comunicações secundárias, existem oito canais ADC de baixa velocidade de 10 *bits* de resolução com leitura por software. A velocidade destes ADCs é de $1,25$MHz com uma largura de 200kHz. Estes canais podem ser utilizados para leitura de sinais de baixa velocidade tais como os fornecidos por uma estação meteorológica ou aparelhos semelhantes. Paralelamente existem oito DACs de 8 *bits* a baixa velocidade. Estes DACs podem ser utilizados para fornecer tensões de controlo, como por exemplo a um motor passo-a-passo que controle a posição de uma antena. A placa mãe apresenta também portas I/O de 64 *bits* de alta velocidade em que 32 *bits* são dedicados à recepção e os outros 32 *bits* à transmissão. Estes bits são controlados por software, tal como o resto do equipamento, através de registos especiais na FPGA.

A FPGA é o modelo Cyclone EP1C12 da Altera. Na figura 4.4 observa-se claramente a importância do papel da FPGA no funcionamento do USRP. A FPGA é a central de comunicação de todos os componentes do sistema: conversores, I/O, USB e placas de expansão. A sua principal função é a conversão da cadeia de amostras para uma taxa mais reduzida de forma a ser enviada através da USB.

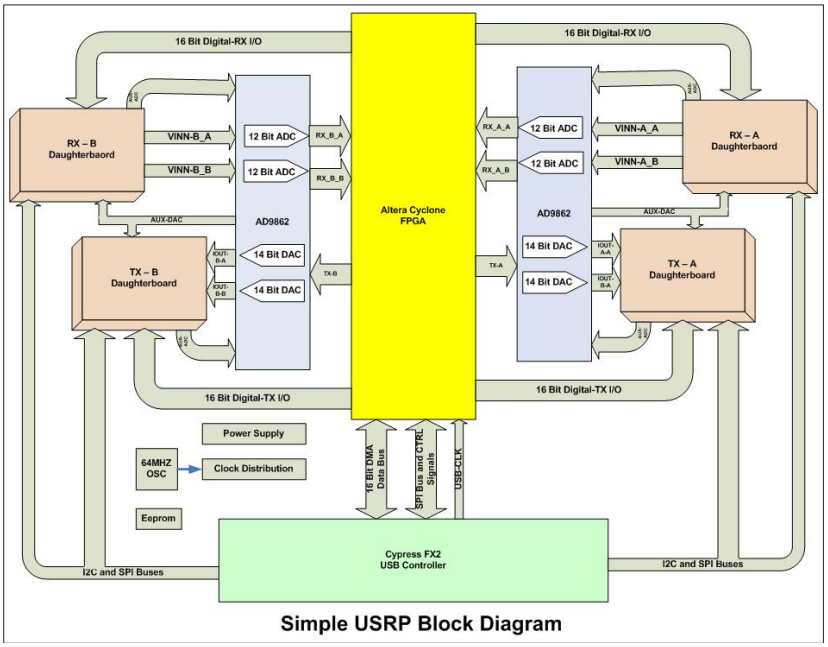


Figura 4.4: USRP - diagrama

O código padrão implementa dois DDCs. O DDC é em parte composto por quatro níveis de filtros integradores combinados em cascata (CIC) realizados apenas com recurso a somadores e atrasos. Para completar, filtros de meia banda com trinta e um coeficientes são postos em série com os filtros CIC. Existe também uma configuração de quatro canais reais em que não são incorporados os filtros de meia banda.

No caso da utilização de quatro DDCs é feita uma amostragem complexa. As componentes I/Q de cada placa de expansão são multiplicadas por uma exponencial complexa de frequência constante. A sinusóide correspondente é gerada através da implementação de um NCO com algoritmo Cordic. Se a frequência utilizada for a IF, o resultado é um sinal complexo em banda base. Este sinal é decimado

por uma sequência composta por um filtro passa-baixo seguido de um decimador. No final a largura de banda máxima da amostragem será inversamente proporcional à taxa de decimação. A utilização destes filtros implica um factor mínimo de 8 e máximo de 256.

Após todos os cálculos, a máxima taxa de transferência constante que se pode garantir é de 32MByte/s. As amostras são enviadas em conjuntos de 32 *bits*, 16 *bits* para a componente I e 16 *bits* para a componente Q. O diagrama deste processo é ilustrado na figura 4.5

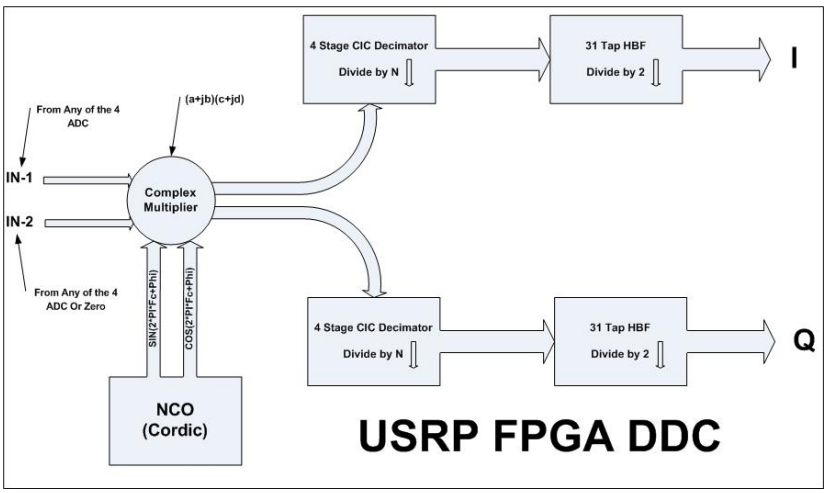


Figura 4.5: USRP - diagrama do DDC

Temos por isso, no máximo, uma largura de banda efectiva de 8MHz usando um factor de decimação igual a 8 e de 250kHz para um factor de 256. Em seguida as amostras são organizadas numa trama com o formato da figura 4.6 e enviadas pela porta USB utilizando o integrado FX2 Cypress.

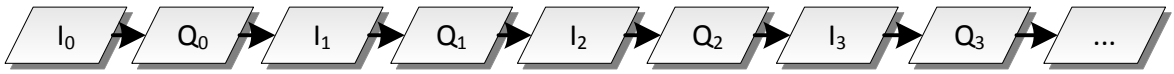


Figura 4.6: USRP - exemplo de uma trama de envio de 4 canais

Por fim esta cadeia é transferida através de USB para o computador utilizando o integrado FX2 da Cypress.

4.3.1.2 *Daughterboards*

Uma das características que torna a plataforma extremamente versátil é o facto de ser possível adicionar placas de expansão. Estas placas permitem a aquisição de uma grande gama de frequências. Podemos separá-las em três grupos: transmissores, receptores e transceptores. Alguns exemplos são apresentados na seguinte lista:

Low Frequency TX/RX (LFRX/LFTX) Possui dois conectores SMA para ligação a fontes externas de sinal ou cargas para transmissão, dependendo do modelo. Funcionam para frequências abaixo dos 30MHz e têm implementado um filtro passa-baixo para eliminar imagens.

TVRX Apenas existe a versão para recepção. Funciona na banda VHF e UHF, de 50 a 860MHz com uma banda de passagem de 6MHz.

RFX São um sistema transceptor com osciladores locais independentes para recepção e transmissão. Apresenta um comutador TX/RX que permite que o sinal seja enviado e transmitido pelo mesmo conector. São disponibilizados 5 modelos que cobrem as bandas desde 400MHz até 2.5GHz.

Este pequeno conjunto demonstra a flexibilidade de análise de espectro deste *hardware*. Embora exista um conjunto de placas de expansão prontas para utilização, devido ao facto de se tratar de *hardware* aberto é possível desenvolver novas placas que encaixem nos *slots* fornecidos para expansão.

Para este trabalho apenas nos iremos focar na placa BasicRX, ilustrada na figura 4.7.

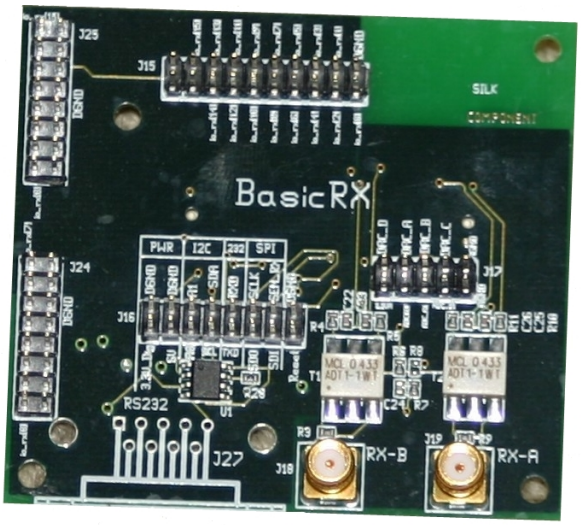


Figura 4.7: USRP - placa de expansão Basic RX

O sinal é aplicado a um máximo de dois conectores SMA, que estão acoplados aos ADCs por transformadores balanceados sem qualquer tipo de misturadores, filtros ou amplificadores.

São idênticas às LFRX/LFTX excepto na gama de funcionamento, sendo indicadas para frequências entre 0.1 e 300MHz. São fornecidas interfaces ainda SPI, I2C e RS232 acessíveis por um conjunto de pinos e muito úteis para controlo de equipamento externo. dezasseis pinos para I/O podem também ser controlados por intermédio da placa. Finalmente, dois ADCs e dois DACs de baixa velocidade são também acessíveis através a BasicRX.

4.3.1.3 *Software*

Sendo uma peça de *hardware* para SDR, o software tem um papel vital na facilidade e flexibilidade do desenvolvimento assim como no seu desempenho. Dito isto, o USRP demonstra ser um dos mais versáteis na hora de implementar soluções complexas.

Começando pelo coração do sistema, o código Verilog que é carregado para a FPGA está disponível para alteração. Deste modo, podem ser realizadas optimizações em função do objectivo do estudo. Os filtros aplicados por defeito podem ser alterados assim como intervenções que maximizem a largura de banda, ou qualquer outro requisito, para o número de entradas utilizadas. Este é um dos pontos de partida para uma solução em que o desempenho seja um dos pontos mais importantes.

Seguindo para o controlo das placas de expansão, os *drivers* básicos são realizados em C++. Esta é uma linguagem que dispensa apresentações, sendo utilizada largamente no mercado pelas suas funcionalidade, flexibilidade e desempenho.

Beneficia também de um relacionamento próximo com o *software* GNU Radio, um conjunto de bibliotecas destinadas ao processamento de sinal. Por este motivo, várias bibliotecas específicas para interface com o USRP são disponibilizadas. Isto permite um progresso rápido e com menos complicações.

4.4 GNU Radio

4.4.1 Introdução

O GNU Radio é um *toolkit* de processamento de sinal disponibilizado de forma gratuita ao abrigo da licença GNU General Public Licence (GPL), derivado do projecto Pspectra que hoje é conhecido pela sua versão comercial – Vanu Software Radio –. Esta licença obriga a que o código utilizado seja aberto a todos os que tiverem interesse em lê-lo ou modificá-lo. Esta é uma mais valia no que respeita ao apoio de desenvolvimento, visto que todo um grupo de programadores profissionais ou aficionados trabalha em conjunto para resolver os problemas da comunidade. O que sem dúvida reduz os custos na medida em que algumas das soluções podem aparecer vindas de pessoas não contratadas que demonstram deste modo o seu interesse pelo projecto. Este facto pode ser comprovado visitando o fórum oficial do projecto [12], onde não só utilizadores colocam e respondem a dúvidas assim como os próprios criadores e responsáveis pela manutenção do projecto o fazem. Entre alguns dos elementos mais importantes encontram-se Eric Blossom (actual responsável), Johnathan Corgan, Firas Abbas Hamza e Matt Ettus.

A plataforma consiste num conjunto de bibliotecas onde estão disponíveis diversas primitivas de processamento de sinal implementadas em C++ [5] e Python [22]. Ambas as linguagens são orientadas a objectos e amplamente utilizadas em sistemas não só de investigação mas também de produtos comerciais. A linguagem C++ dispensa introduções visto que se trata de uma linguagem antiga com uma forte presença no mercado e com uma grande quantidade de documentação para os mais diversos sistemas. A linguagem Python tem vindo a ganhar força na indústria ao longo dos anos, sendo nos dias de hoje um grande apoio a distribuições de sistemas operativos (SO) baseados em LINUX assim como por gigantes da Internet como é o caso da Google. É uma linguagem cujo paradigma assenta na funcionalidade, orientação a objectos e um tipo de escrita dinâmica e forte. Isto é, não são necessárias declarações de tipos de variáveis, à semelhança do MATLAB, e essas mesmas variáveis podem alterar de tipo ao longo do programa. O facto de ter vindo a crescer deu origem a um forte apoio na documentação, criação de bibliotecas e de paradigmas de programação em Python.

A existência uma grande variedade de fontes de ajuda é um ponto essencial no que respeita à facilidade de aprendizagem, depuração e concretização de uma ferramenta desde a raiz para ser aplicada em campo, como é o caso desta tese.

Dito isto, é agora necessário conhecer os princípios básicos de funcionamento da arquitectura da plataforma GNU Radio assim como os sistemas disponíveis para a sua utilização. Estes dois pontos são cobertos pelos tópicos 4.4.2 e 4.4.3 desta secção.

4.4.2 Arquitectura

4.4.2.1 Estrutura e paradigmas de um diagrama

A plataforma GNU Radio assenta sobre as linguagens Python e C++. É por isso necessário encontrar uma forma de interligar ambas as linguagens de forma a que a sua utilização seja transparente ao chamar funções entre elas. Com isto queremos dizer que é preciso um código *cola* que permite ao ambiente Python chamar funções implementadas em código C++ passando ele próprio os parâmetros para os cabeçalhos. Este código *cola* trata-se do projecto Simplified Wrapper and Interface Generator (SWIG). Para que seja implementado é preciso criar um ficheiro adicional `*.i` onde são incluídos os cabeçalhos das funções em C++ e as bibliotecas SWIG que farão a tradução entre linguagens. Esta aproximação permite, não apenas neste projecto mas em muitos outros casos, aproveitar o melhor de vários mundos, pois muitas vezes apercebe-mo-nos de que determinada linguagem é ideal para implementar uma dada tarefa mas que não tem um desempenho tão bom noutra tarefa do mesmo sistema. A utilização desta técnica permite por exemplo usar linguagens de alto desempenho (mais

complexas) em tarefas críticas e linguagens de *scripting* (menos complexas) na gestão de recursos e na validação e controlo de ciclos.

Depois desta explicação, podemos fazer o paralelo para a arquitectura do GNU Radio. Por um lado, a linguagem C++ é conhecida principalmente pelas seguintes razões: elevado desempenho, flexibilidade e complexidade. Por outro lado, a linguagem Python é conhecida por: desempenho muito inferior a C++, flexibilidade e funcionalidade. São por isso duas linguagens que se complementam e por este motivo tornam-se a escolha ideal para a implementação desta arquitectura.

Podemos subdividir as tarefas em dois grandes grupos: críticas e não críticas. As críticas são aquelas em que a velocidade de cálculo é o factor principal. As não críticas são aquelas em que a capacidade de cálculo a grande velocidade não é vital. Neste ponto podemos desde já inferir que as tarefas críticas são implementadas em C++ e as não críticas em Python.

Para compreendermos quais são as tarefas críticas e não críticas precisamos avançar mais em detalhe no modo de funcionamento do GNU Radio. À semelhança do Simulink da MathWorks, o objectivo é criar um diagrama de fluxo de sinal que conduz as nossas amostras do ponto A até ao ponto B efectuando todas as transformações necessárias ao longo do seu trajecto. Para melhor ilustrar o paradigma, é apresentado na figura 4.8 um diagrama da implementação de um receptor de rádio FM. Cada bloco representa um tipo de transformação aplicada ao sinal, enquanto que as setas indicam o caminho sequencial percorrido por este. Embora não seja explícito através da observação da figura, este conjunto de passos é executado ciclicamente, parando apenas quando alguma anomalia a nível de equipamento/código ocorra ou se o utilizador assim o entender.

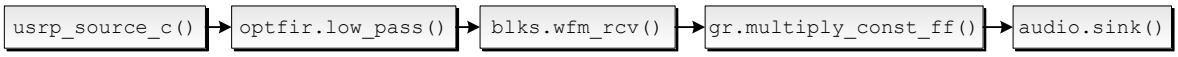


Figura 4.8: GNU Radio - diagrama de um rádio FM

Uma breve observação do processo resulta na seguinte descrição:

1. O sinal de origem é adquirido pelo USRP e enviado em forma de *stream*;
2. É aplicado um filtro para eliminar o ruído em torno da frequência sintonizada;
3. É usada uma função que efectua a desmodulação FM para o seu formato áudio original;
4. O resultado da desmodulação é multiplicado por uma constante de modo a controlar o volume sonoro;
5. A stream desmodulada é multiplicada e enviada para a placa de som.

As tarefas críticas são geralmente as transformações desempenhadas nos blocos, enquanto que as não críticas são o controlo de fluxo de dados entre blocos e validação das variáveis de controlo.

Pelo facto de os blocos representarem a transformação do sinal, uma grande capacidade de cálculo é-lhes exigida para que não ocorram perdas de sincronismo com os fenómenos que estão a ser observados levando a falhas de correlação dos dados ao longo do tempo. Além da possibilidade de perda de correlacionamento com o sinal observado surge também o risco de perda de sincronismo com outras análises paralelas que podem depender do resultado para prosseguirem. Estes são os dois motivos principais que tornam os blocos um elemento crítico do diagrama.

No entanto o controlo de fluxo de dados entre blocos não requer uma elevada capacidade de cálculo, sendo em geral apenas um conjunto de validações e escolha de parâmetros de cálculo e não o cálculo em si, assim como a indicação de quais os blocos que se encontram interligados. Ambas as tarefas não são computacionalmente exigentes, e por isso a utilização de uma linguagem de muito

alto nível, como é o caso do Python, adequa-se a este processo. Desta forma o código necessário para implementar os algoritmos de controlo é em grande parte simplificado pela utilização de *scripting*.

Um exemplo simples do código Python que implementa o controlo de um sinal sinusoidal é apresentado em seguida:

gnuradio_hello_world.py

```
 1:    def build_graph():
 2:       fa = 48000
 3:       f = 350
 4:       A = 1
 5:       fg = gr.flow_graph()
 6:       src = gr.sig_source_f(fa, gr.GR_SIN_WAVE, f, A)
 7:       dst = audio.sink(fa)
 8:       fg.connect ((src, 0), (dst, 0))
 9:       return fg
10:
11:    if __name__ == '__main__':
12:        fg = build_graph()
13:        fg.start()
14:        raw_input('Press Enter to quit: ')
15:        fg.stop()
```

As primeiras dez linhas definem um diagrama constituído por 2 blocos: uma fonte – gerador de onda sinusoidal; e um destino – placa de som do sistema. Nas linhas dois a quatro dita-se a frequência de amostragem (`fa`), a frequência do sinal (`f`) e a amplitude do sinal (`A`). Em seguida cria-se a variável `fg`, cujo conteúdo é o objecto `flowgraph` do módulo `gr` pertencente às bibliotecas fornecidas pelo GNU Radio. Este objecto armazena as informações sobre a forma como estão ligados os blocos do sistema. Nas linhas seis e sete são definidos a fonte e o destino. Sendo que a fonte é o objecto `sig_source_f`, também do módulo `gr`. Este objecto emula um gerador de sinal sinusoidal do tipo *float* com os parâmetros de frequência e amplitude estabelecidos anteriormente. O destino é a placa de som, representada pelo objecto `sink` do módulo `audio`, também este fornecido pelo GNU Radio. Por fim indica-se ao objecto `fg` que o bloco do gerador de sinal está ligado à placa de som.

Da linha onze em diante é feita uma verificação para confirmar se o código é chamado como programa principal. Se o for, é criada a variável que contém o diagrama de fluxo de sinal definido anteriormente e é dada a ordem de arranque. Fica depois em funcionamento até que seja pressionada a tecla que irá despoletar o processo de paragem do diagrama.

Podemos observar neste exemplo tudo o que foi dito sobre a arquitectura GNU Radio até agora:

- As funções `gr.flowgraph`, `gr.sig_source_f(sample_freq,type,freq,amp)`, `audio.sink(sample_freq)`, `fg.connect`, `fg.start` e `fg.stop` são implementadas em C++, mas são chamadas pelo código Python através da passagem de parâmetros ao seu cabeçalho original;
- O paradigma do diagrama é implementado através do objecto `gr.flowgraph()` que permite armazenar informação sobre a ligação entre os blocos criados;
- As funções que implicam cálculo de alta velocidade enunciadas no primeiro ponto são implementadas em C++;
- O controlo do diagrama é implementado em Python, na medida em que as ordens de arranque, paragem, ligação entre blocos e parâmetros são descritos nesta linguagem.

O exemplo apresentado na figura 4.8, em conjunto com o código representa de uma forma simples, respectivamente, a parte conceptual e prática do paradigma de um sistema desenvolvido com a plataforma GNU Radio.

Como complemento aos dois exemplos, apresentamos na figura 4.9 uma interface criada com as bibliotecas WxPython para o sistema da figura 4.8. Para mais detalhes e paradigmas desta biblioteca recomendamos a leitura do livro “WxPython in action” [44]

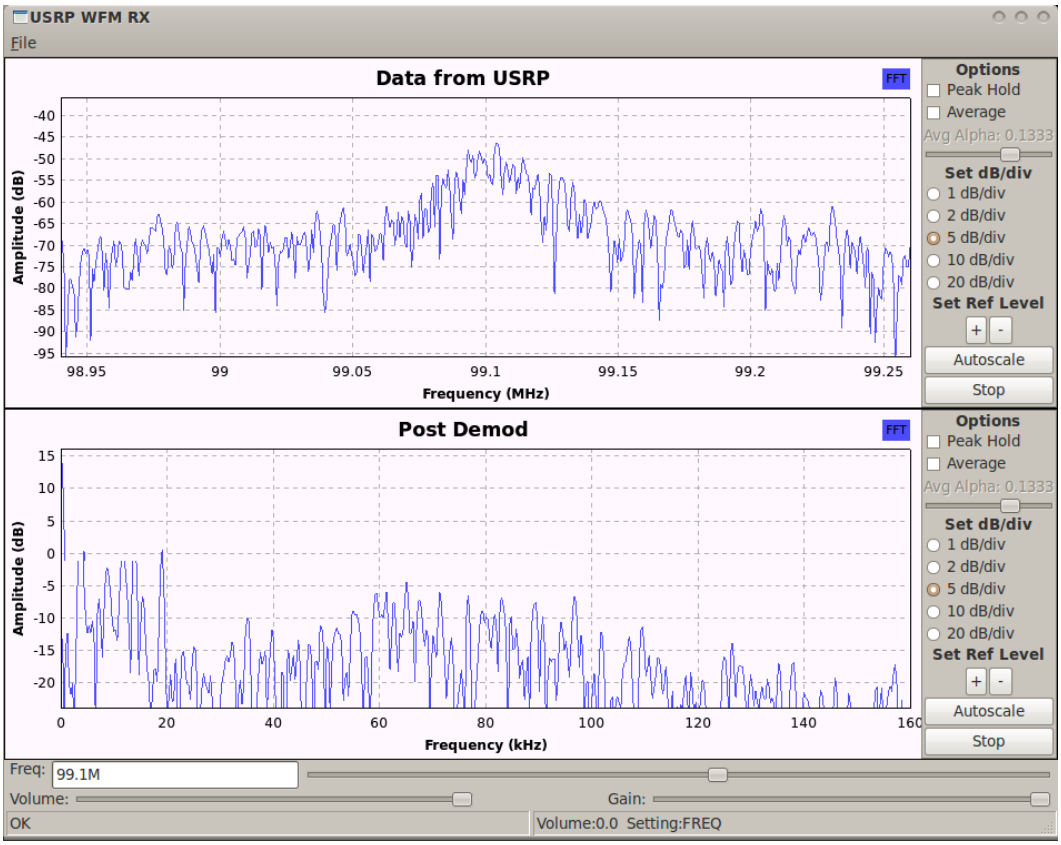


Figura 4.9: GNU Radio - interface do rádio FM

Actualmente esta plataforma oferece de raiz várias primitivas, conhecidas por blocos. Tais incluem suporte ao *hardware* USRP, acesso a dispositivos de som compatíveis com os *drivers* ALSA e OSS e apresentação de interfaces gráficas usando wxPython [26] e QT [27]. Alguns exemplos de funções presentes no pacote de instalação incluem:

- Desmodulação AM;
- Codificação BPSK e QPSK;
- Modulação e desmodulação GMSK;
- Transmissor e Receptor FM em banda estreita e banda larga;
- Gerador de CRC;
- Configuração TCP/UDP;
- Exemplos de aplicações básicas de análise de FFT;

Uma interface semelhante ao Simulink da MathWorks está também disponível. Chama-se GNU Radio Companion (GRC) e permite conectar blocos e visualizar resultados. Embora não seja tão versátil como a realização de sistemas através de código tem duas vantagens. A primeira é facilitar a compreensão do funcionamento do GNU Radio ao principiante. A segunda é a rapidez com que

é possível testar um bloco criado sem a necessidade de recorrer a programação, que é sempre um processo mais moroso. Uma ilustração da interface é apresentada na figura 4.10.

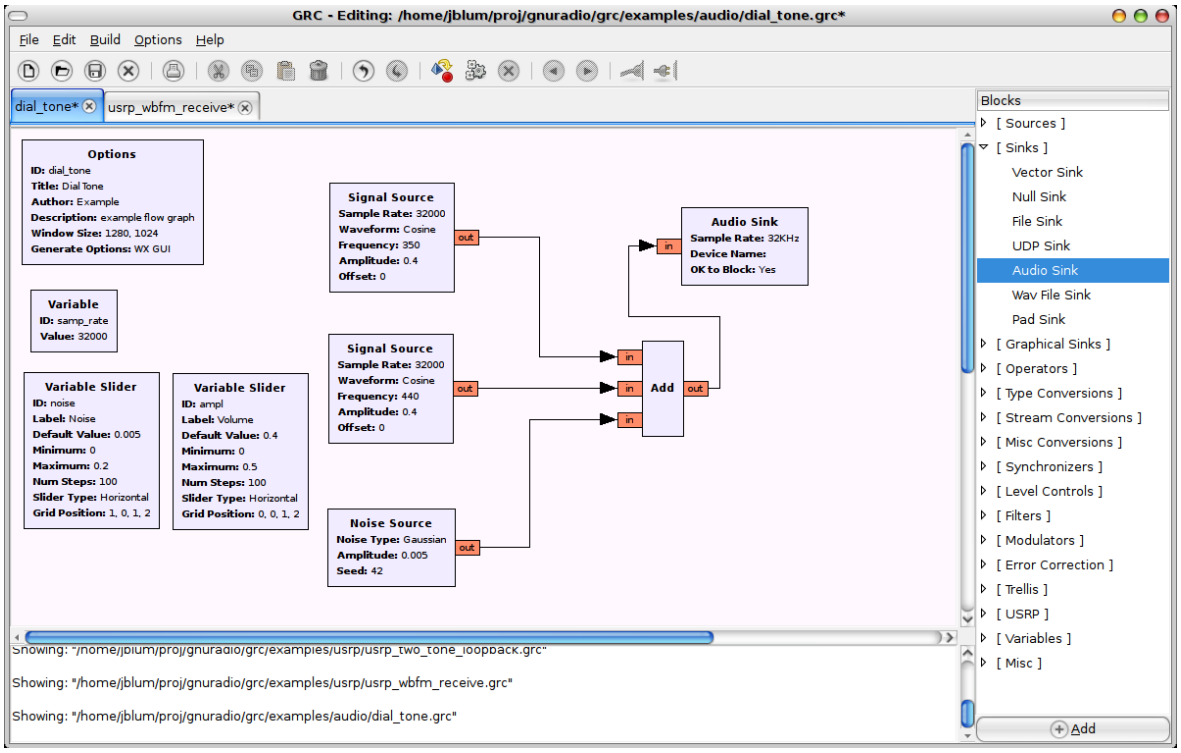


Figura 4.10: GNU Radio Companion - exemplo de DTMF

4.4.2.2 Estrutura de um bloco

Resta então saber a estrutura de um bloco e o que fazer para criar um. Tal como foi demonstrado na figura 4.8 um bloco pode pertencer a uma de três categorias: fonte, destino ou de passagem.

No caso de se tratar de um bloco fonte apenas existem saídas. O número de saídas é predefinido em código através da atribuição de um valor mínimo e máximo. Associado a cada saída está o seu tipo. Este pode ser um valor inteiro, vírgula flutuante, complexo ou qualquer outro tipo existente em C++. Se for um bloco de destino são definidas apenas as entradas. O processo de criação de entradas é em tudo semelhante ao de criação de saídas, apenas diferindo no facto de receber dados ao invés de enviar. Um bloco de passagem nada mais é que uma mistura de ambos os anteriores. Possui por isso um número mínimo e máximo predefinido para entradas e saídas do bloco. É de registar o facto que o número mínimo e máximo pode ser coincidente.

Neste tópico vamos passar a explicar a estrutura básica do código que define o bloco. Visto que este é criado em C++, a utilização de classes é indispensável para simplificar o código permitindo ao mesmo tempo a adição de novas funcionalidades mais rapidamente.

Os novos blocos foram criados com recurso aos exemplos *How to Write a Signal Processing Block* [15] criados por Eric Blossom. Com base nestes exemplos apenas 3 ficheiros têm que ser alterados para criar a nossa própria biblioteca: `howto_*.h`, `howto_*.cc` e `howto.i`.

Os ficheiros fonte `howto_*.h` contêm os cabeçalhos que definem as classes que representam os nossos blocos. O modelo básico do código é apresentado em seguida:

`howto_bloco.h`

```
01: #ifndef INCLUDED_HOWTO_BLOCO_H
02: #define INCLUDED_HOWTO_BLOCO_H
```

```
03:
04: #include <gr_block.h>
05:
06: class howto_bloco;
07:
08: typedef boost::shared_ptr<howto_bloco_ff> howto_bloco_sptr;
09:
10: howto_bloco_sptr howto_make_bloco ();
11:
12: class howto_bloco : public gr_block
13: {
14:   private:
15:
16:   friend howto_bloco_sptr howto_make_bloco ();
17:
18:   howto_bloco ();
19:
20:   public:
21:     ~howto_bloco ();
22:
23:     int general_work (int noutput_items,
24:                       gr_vector_int &ninput_items,
25:                       gr_vector_const_void_star &input_items,
26:                       gr_vector_void_star &output_items);
27: };
28: #endif /* INCLUDED_HOWTO_BLOCO_H */
```

As linhas 1, 2 e 28 servem apenas para confirmar que este bloco não existe em duplicado na altura da compilação. Na linha 4 é incluída a classe progenitora da classe que estamos a criar. Isto é, todas as propriedades, variáveis e métodos que esta possui são herdadas pela nossa classe. Isto impede que seja necessário definir todos os métodos novamente, de modo a manter a coesão entre todas as classes criadas pela plataforma GNU Radio. As linhas 8 e 10 demonstram a utilização das bibliotecas Boost ao serviço do GNU Radio. Deste modo são criados ponteiros inteligentes que podem ser utilizados pelo SWIG para fazer a interface entre o C++ e o Python e que ao mesmo tempo fazem a gestão de memória devido à forma como foram implementados. O conteúdo das linhas 12 a 27 é a definição da nossa classe. Geralmente entre os elementos privados constam as variáveis ou funções que apenas têm interesse ao nível de funcionamento interno da classe ou aquelas que o programador não deve ser capaz de alterar. Alguns exemplos são o constructor, o ponteiro para a classe e outras variáveis como contadores *et cetera*. O destructor é sempre uma função de acesso público assim como a `general_work(...)`. A `general_work(...)` é a zona de maior interesse do bloco, visto que é a tarefa que o bloco realizará. Os quatro parâmetros de entrada desta função são sempre:

- O número de valores na saída (`noutput_items`) que nos indica o tamanho do vector de saída, pois além de uma *stream* de valores podemos também ter uma *stream* de vectores;
- Os valores que constituem cada entrada (`&ninput_items`);
- Dois ponteiros para os vectores de entrada e saída (`&input_items` e `&output_items`), para que seja possível ler os elementos da listas recebidas e escrever os resultados nas listas de saída.

Isto conclui a visão geral da construção de um cabeçalho de um bloco na arquitectura GNU Radio.

Com o que foi dito em mente passamos à explicação de como escrever os ficheiros `howto_*.cc` onde se encontra o código que implementa o trabalho útil que o bloco irá realizar. Em seguida é apresentado o código base para esta implementação:

---

howto_bloco.cc

```
01: #ifdef HAVE_CONFIG_H
```

```
02: #include "config.h"
03: #endif
04:
05: #include <howto_bloco.h>
06: #include <gr_io_signature.h>
07:
08: howto_bloco_sptr howto_make_bloco ()
09: {
10:   return howto_bloco_sptr (new howto_bloco ());
11: }
12:
13: static const int MIN_IN = 1;
14: static const int MAX_IN = 1;
15: static const int MIN_OUT = 1;
16: static const int MAX_OUT = 1;
16:
18: howto_bloco::howto_bloco ():gr_block ("bloco",
19:                              gr_make_io_signature (MIN_IN, MAX_IN, sizeof (float)),
20:                              gr_make_io_signature (MIN_OUT, MAX_OUT, sizeof (float)))
21: {
22: }
23:
24: howto_bloco::~howto_bloco ()
25: {
26: }
27:
28: int howto_bloco::general_work (int noutput_items,
29:                                gr_vector_int &ninput_items,
30:                                gr_vector_const_void_star &input_items,
32:                                gr_vector_void_star &output_items)
33: {
34:   /* !!! Criar função aqui !!! */
35:
36:   consume_each (noutput_items);
37:   return noutput_items;
38: }
```

Nas linhas 5 e 6 são incluídas as bibliotecas básicas: a que define a nossa classe e a que define a manipulação de entradas/saídas nos blocos do GNU Radio. Entre a linha 8 e 11 é criado o ponteiro inteligente, da biblioteca Boost, que irá indicar ao nosso programa a localização da nossa classe em memória. As quatro definições seguintes indicam os limites máximos e mínimos tanto para as entradas como para as saídas. Em seguida está escrito o construtor que neste caso não efectua nenhuma operação em especial, além do que foi herdado do bloco `gr_block`, e por esse motivo encontra-se vazio. Também o destrutor nas linhas 24, 25 e 26 se encontra vazio pelas mesmas razões. Entre as linhas 28 e 38 encontra-se o coração do bloco. Sendo, em geral, neste espaço que toda a acção acontece. Aqui é definido o processo realizado pelo bloco. Opcionalmente podem ser incluídos métodos que efectuam operações adicionais tais como extracção de dados do bloco ou alteração de parâmetros utilizados por este de uma forma mais segura. Estes métodos são definidos depois das linhas apresentadas da seguinte maneira: "`tipo howto_bloco::metodo(...){...}`"; e o seu cabeçalho deve ser incluído nas variáveis públicas no ficheiro `howto_bloco.h` da seguinte forma: "`tipo howto_bloco::metodo(...);`". Sem entrar em pormenor, o conteúdo entre parêntesis são as variáveis de entrada do método e entre chavetas deve ser posto o algoritmo.

Agora que sabemos como definir e construir o nosso algoritmo temos que criar o ficheiro `howto.i` que será o utilizado pelo SWIG para realizar a interface entre o Python e o código C++ que acabamos de escrever. As seguintes linhas apresentam o conteúdo que este ficheiro deve ter para os dois exemplos utilizados neste tópico:

---

howto.i

```
01: %include "exception.i"
02: %import "gnuradio.i"
```

```
03:
04: %{
05:   #include "gnuradio_swig_bug_workaround.h"
06:   #include "howto_bloco.h"
07:   #include <stdexcept>
08: %}
09:
10: GR_SWIG_BLOCK_MAGIC(howto,bloco);
11:
12: howto_bloco_sptr howto_make_bloco ();
13:
14: class howto_bloco : public gr_block
15: {
16:  private:
17:    howto_bloco ();
18: };
```

As primeiras oito linhas, com excepção da 6, importam ficheiros comuns a todas as implementações do SWIG no GNU Radio. A sexta linha importa o ficheiro `*.h` criado para o nosso bloco. Na linha 10 é indicado o nome do módulo no qual será incluído o nosso bloco, neste caso `howto`. Em seguida é incluído o ponteiro criado pelo Boost. Finalmente indicamos o nome da classe criada e o cabeçalho de todos os métodos que pretendemos chamar através do Python, neste caso apenas o construtor está definido.

Para terminar este processo resta apenas um passo, compilar. Para tal temos que alterar o ficheiro `Makefile.am`, fornecido com os exemplos de Eric Blossom, e substituir o nome dos ficheiros para os criados por nós: `howto_bloco.h` e `howto_bloco.cc`. Neste momento ao executarmos os comandos `make` e `make install` no terminal tudo será compilado e ficará pronto para incluir no código Python. Para isso basta incluir uma linha com o seguinte código: `import howto`.

Isto conclui uma visão mais aprofundada sobre a arquitectura e criação de programas e blocos na plataforma GNU Radio. Mais detalhes serão apresentados durante a revisão dos algoritmos implementados no receptor de que é alvo esta dissertação.

4.4.3 Instalação

Para podermos utilizar as bibliotecas GNU Radio é necessário primeiro proceder a uma instalação. Como acontece com qualquer outro *software*, este procedimento levanta vários tipos de problemas. Entre eles encontram-se a aquisição do *software* propriamente dito, o *hardware* compatível, o sistema operativo sobre o qual funciona e finalmente as ferramentas de *software* adicionais que compõem o ambiente de programação.

No que diz respeito à aquisição, é o mais simples de resolver. Pelo facto de ser distribuído de forma gratuita, uma visita à página oficial do projecto [11] indicará todas as formas de fazer o *download* dos ficheiros necessários.

O problema relacionado com o *hardware* é também de fácil resolução, na medida em que esta plataforma foi desenvolvida de forma a poder funcionar com uma vasta gama de processadores disponíveis no mercado. É compatível desde processadores comuns, como é o caso dos fabricados pela Intel e AMD, até a alguns processadores mais exóticos como é o caso do Cell Broadband Engine instalado na Playstation 3. No caso de se utilizar uma placa de som, o único requisito é ser compatível com os drivers ALSA ou OSS.

A resposta à questão do sistema operativo é mais complexa, mas não mais difícil de resolver que as anteriores. Também neste aspecto o GNU Radio demonstra uma grande capacidade de adaptação. Praticamente qualquer sistema Windows ou baseado em UNIX pode ser utilizado. Embora a plataforma Windows seja compatível, um emulador de LINUX tem que ser instalado e que serve de plataforma de

lançamento dos programas criados. Tanto o Cygwin [10] como o MinGW [18] podem ser utilizados. Por este motivo as preferências da comunidade recaem em sistemas operativos baseados em UNIX. Com base neste requisito, os sistemas com o *kernel* LINUX são de longe os mais utilizados. De entre todas as distribuições distinguimos principalmente as seguintes: Ubuntu [25] e Fedora [9]. Várias outras distribuições são utilizadas sem qualquer problema de compatibilidade, como é o caso da CentOS [4], openSUSE [21], Debian [6] *et cetera*. Outras opções com bastantes adeptos são SOs baseados em BSD, tais como OSX [2] da Apple ou o NetBSD [19].

Finalmente temos que garantir que o ambiente de programação se encontra completo. Para isso falta a instalação das ferramentas de *software* adicionais que são requeridas pelos blocos implementados no pacote GNU Radio, tais como compiladores, interpretadores, editores e outras bibliotecas utilizadas pelas funções da plataforma. Novamente, na página oficial do projecto está disponível um guia para estabelecer o ambiente de trabalho.

Em seguida vamos descrever e justificar as escolhas feitas acerca dos quatro pontos explicados anteriormente.

As características de interesse do *hardware* disponível para a realização desta tese são as seguintes:

Portátil
Toshiba A200 PSAE6
CPU Intel Core2 Duo T5500@1.83GHz
RAM 3GB DDR2@667MHz
Chipset Intel® 945PM Express

PC
Marca não definida
CPU Intel Core2 Duo E6600@2.40GHz
RAM 2GB DDR2@800MHz
Chipset Intel® P35

Embora tenham sido utilizados dois computadores, apenas um é necessário. O portátil destina-se a desenvolvimento fora do laboratório, enquanto que o PC é a máquina efectiva. É de notar que nenhuma alteração no código é necessária entre máquinas.

Sendo o nosso objectivo o desenvolvimento de um detector de baixo custo, o software é o primeiro ponto onde podemos reduzir o orçamento. Ao utilizar uma solução distribuída de forma aberta e gratuita efectuamos a primeira poupança, não tendo que pagar licenças. Neste caso a solução encontrada foi a plataforma GNU Radio, pois trata-se de um pacote de *software* muito utilizado na área de processamento digital de sinal. E por este motivo tem sido testada exaustivamente e mantém-se em constante evolução ao mesmo tempo que dispõe do apoio de uma comunidade interessada em fazer a diferença no avanço de uma tecnologia emergente e comum a tantas áreas da comunicação.

O sistema operativo escolhido foi a distribuição Ubuntu com o *kernel* LINUX. Outra vez a escolha recai em *software* aberto e distribuído gratuitamente. Neste caso além do factor custo outros foram considerados. Em primeiro lugar a utilização de um emulador de LINUX não é aconselhada quando se tem acesso a um sistema LINUX nativo. Em segundo lugar, embora existam dezenas de distribuições gratuitas de LINUX, esta trata-se da mais adoptada comercialmente. E por esse facto a documentação, apoio e outras ferramentas disponíveis existem em maior quantidade. Outro ponto importante é a facilidade de instalação e utilização. Como foi mencionado anteriormente, são dois factores que agilizam o desenvolvimento, reduzindo os custos indirectamente como desejado. Finalmente o processo de instalação, não do SO mas sim da plataforma GNU Radio, está bastante optimizada para este sistema. Todas estas razões tornam-no um dos sistemas ideais para realizar este projecto. Outras opções que merecem menção são as distribuições Fedora e Debian, esta última pode ser considerada a *mãe* do Ubuntu.

Quanto a ferramentas adicionais um guia está publicado no site oficial do GNU Radio [13]. Em seguida é apresentada a lista de requisitos para uma correcta instalação da plataforma GNU Radio:

Ferramentas de Desenvolvimento

- g++
- subversion
- make
- autoconf, automake, libtool
- sdcc (do "universe"; 2.4 ou mais recente)
- guile (1.6 ou mais recente)
- ccache

Bibliotecas

- python-dev
- FFTW 3.X (fftw3, fftw3-dev)
- cppunit (libcppunit and libcppunit-dev)
- Boost 1.35 (ou mais recente)
- libusb and libusb-dev
- wxWidgets (wx-common) e wxPython (python-wxgtk2.8)
- python-numpy (via python-numpy-ext)
- ALSA (alsa-base, libasound2 and libasound2-dev)
- Qt (libqt3-mt-dev para versões mais recentes que 8.04; versão 4 funciona para 8.04 e seguintes)
- SDL (libsdl-dev)
- GSL GNU Scientific Library (libgsl0-dev >= 1.10 necessário para o ramo SVN)

SWIG

- usar o pacote de instalação standard

As instruções apresentadas em [13] servem apenas de guia. Como tal apresentamos no anexo A a sequência de passos efectuada para reproduzir o nosso ambiente de programação.

É de notar que esta é apenas uma das formas de instalar a plataforma. No nosso caso foi a que demonstrou ser mais estável e completa, não apresentando qualquer problema em compilações futuras.

No final de todo o processo de instalação estamos finalmente aptos a começar o ataque ao problema, que é a concepção de um detector inteligente de baixo custo para um sinal de satélite utilizando o USRP em conjunto com a plataforma GNU Radio.

Capítulo 5

Implementação do Receptor Digital

Antes de mais é necessário relembrar o nosso objectivo.

Tal como foi mencionado no primeiro capítulo, será lançado pela ESA o módulo TDP5 a bordo do satélite AlphaSat. Este fará a transmissão de um sinal na banda Q-V, com uma frequência central de aproximadamente 40GHz, com o propósito de ser monitorizado a fim de caracterizar esta zona do espectro para telecomunicações.

Devido à frequência do sinal, tem que ser utilizado um *frontend* RF . Pelo facto da banda do sinal ser estreita, na ordem das dezenas de Hertz, o *frontend* aplica um filtro também ele estreito. Isto faz com que seja necessário ter um cuidado especial com a sintonização de modo a manter o sinal dentro da banda de passagem. Por este motivo o receptor digital necessita de ajustar a frequência utilizada pelo *frontend* para converter a IF intermédia de 2GHz para 10.7MHz. Este ajuste requer a implementação de um canal de comunicação entre o USRP e o equipamento externo. Após a configuração inicial o sinal deve estar localizado dentro da banda do filtro de cristal do *frontend*.

À entrada espera-se uma CNR em torno dos 55dB/Hz e uma potência máxima perto dos 10dBm. A partir do momento que o sinal entra no USRP é iniciada a conversão para digital. Posteriormente é feita uma análise espectral através de algoritmos executados no computador.

Os resultados são usados como referência para o controlo da sintonização tanto ao nível das *daughterboards* do USRP como da PLL do *frontend* RF. Todos os resultados relevantes da análise são guardados continuamente em ficheiros com uma duração máxima de 6 horas e fechados ao final de cada dia independentemente da duração.

5.1 Visão geral

Uma visão geral do sistema pode ser vista na figura 5.1. No que diz respeito a esta dissertação apenas abordaremos as questões do sistema relacionadas com os blocos à direita da IF de 10.7MHz. A única excepção consiste na comunicação SPI com unidade de síntese de um oscilador local utilizado para obter a IF à entrada do USRP. Este e o restante *hardware* do *frontend* foi desenvolvido e implementado numa tese paralela [28], realizada por Emanuel Cabral, que vêm complementar a necessidade de um amplificador e conversor de frequência para uma correcta digitalização do sinal.

Como se pode observar, são adquiridos dois sinais distintos. O *copolar* é o sinal de teste originalmente enviado pelo TDP5, enquanto que o *crosspolar* se trata da sua versão despolarizada o qual surge durante a travessia da frente onda pela atmosfera terrestre.

Visto que os fenómenos meteorológicos estão estreitamente interligados com este acontecimento, é de extrema importância a sua correlação com os dados capturados. Por este motivo o receptor deve fornecer as infraestruturas que permitam integrar dados provenientes de uma estação meteorológica tanto na interface com o utilizador como nos registos binários.

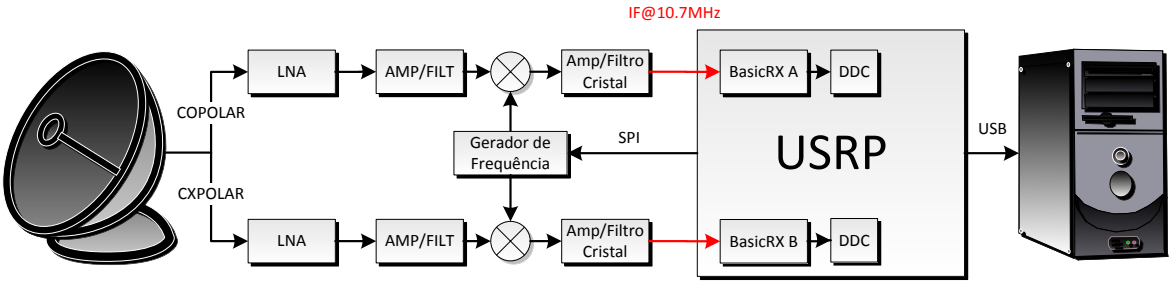


Figura 5.1: Visão geral do receptor digital - *Hardware*

Ambos os sinais sofrem o mesmo processo de filtragem e amplificação de um ponto de vista conceptual, no entanto não percorrem o mesmo caminho eléctrico. Por essa razão deve ser tido em conta que pequenos ajustes ao *frontend* poderão ser necessários de modo a assegurar que a mesma amplificação e conversão de frequência é aplicada. De outra forma os dados capturados podem ser incorrectamente interpretados devido a erros sistemáticos não quantificados no sistema. A ligação entre os circuitos de amplificação e as placas BasicRX no USRP é feita através de cabos com conectores SMA.

Assim que o sinal entra no USRP começa a sua conversão para o domínio digital. A FPGA efectua um processo de DDC de modo a obter o espectro do sinal em banda base. O USRP encontra-se ligado através de USB com o computador, para o qual é enviada uma stream com os dados resultantes do tratamento digital aplicado pela FPGA.

A partir deste ponto a análise é totalmente digital.

O primeiro passo no mundo digital é estabelecer a sequência de transformações e acções de controlo a efectuar. Um diagrama de fluxo simplificado de todo o processo de análise, captura e seguimento é apresentado na figura 5.2, em que cada bloco representa conceptualmente um conjunto de análises e decisões tomadas num dado ponto do processo global. No canto inferior esquerdo de cada um são explicitados os ficheiros onde está implementado o código que realiza a sua tarefa. Estes são explicados em maior detalhe nos tópicos 5.2 a 5.5.

Em traços gerais uma análise do diagrama resulta no seguinte conjunto de operações:

1. É iniciada a interface com o utilizador. Nesta são escolhidos os parâmetros utilizados para a análise e no final dá-se início ao processo de aquisição. Alternativamente, o utilizador não interage com o programa durante um período superior a 10 minutos e então ele arranca automaticamente com a última configuração utilizada;

2. Antes que algo seja feito, é configurada a frequência de conversão por defeito, 2.0107GHz;

3. Começa o processo de análise espectral e validação de sintonia. Assim que o factor de qualidade atinja o valor desejado esta fase é dada como terminada e a frequência descoberta é usada para começar o próximo passo;

4. É feita a aquisição de sinal utilizando a frequência de sintonia descoberta e guardados os ficheiros binários. Utilizando os resultados da análise espectral é avaliada a possibilidade tanto de um ajusto fino (BasicRX) como de um ajusto grosso (PLL);

5. Se for justificável é realizado o ajuste e recomeça do passo anterior. Caso não seja, pode dever-se a duas razões: frequência estável ou CNR baixa. No primeiro caso, o processo segue normalmente. No segundo caso, a actualização de frequência é estagnada por um período mínimo de um minuto enquanto a CNR não recuperar o nível esperado;

6. Recomeçar a partir do passo 4.

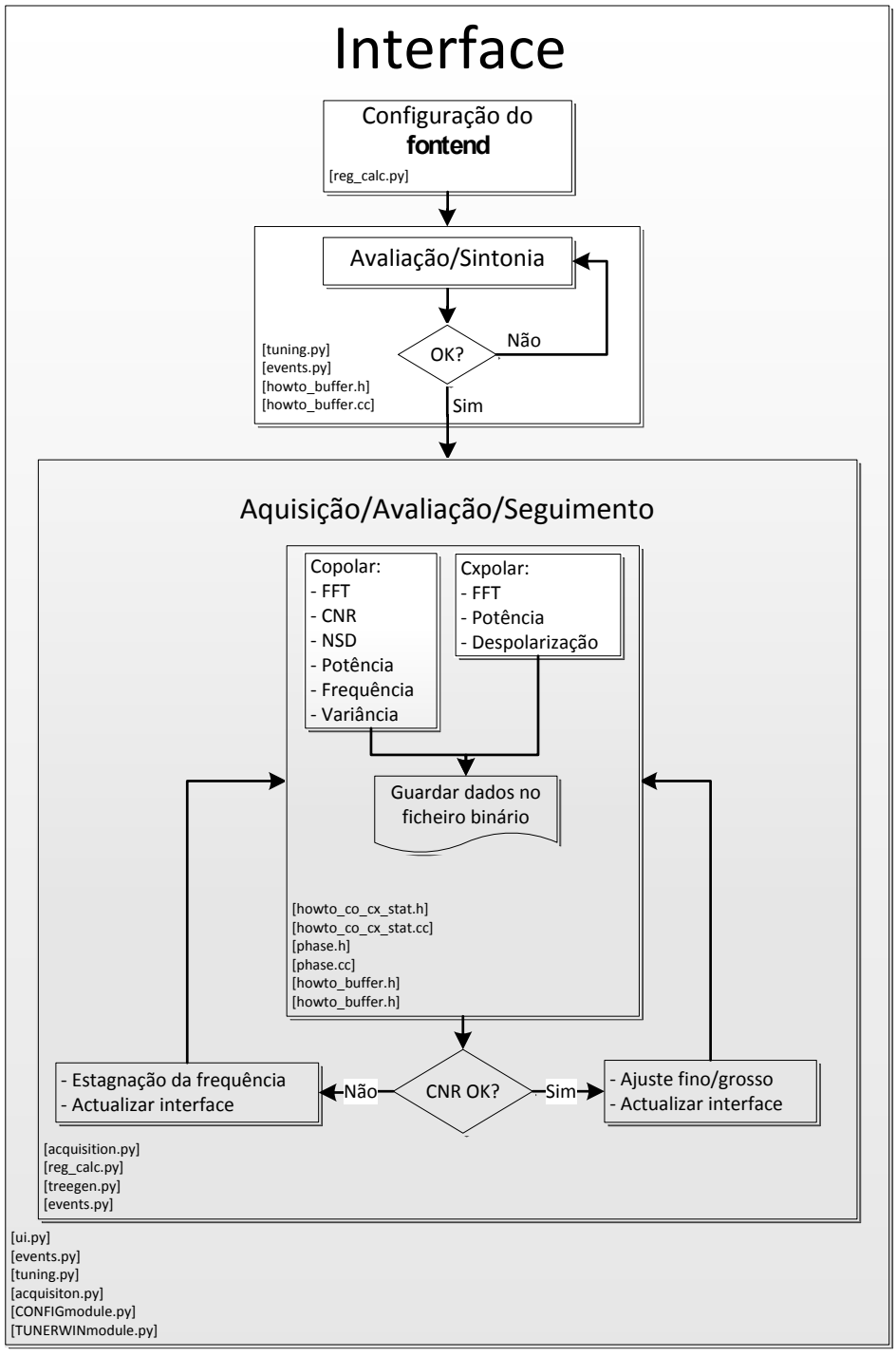


Figura 5.2: Visão geral do receptor digital - *Software*

5.2 Interface

A interface (GUI), embora não seja um elemento essencial do receptor, permite que este se torne uma ferramenta mais poderosa. Isto deve-se principalmente a dois factores, sendo o primeiro

a facilidade da configuração dos parâmetros utilizados para a sintonização e análise do sinal. A utilização de menus gráficos reduz o tempo e complexidade da alteração pois trata-se de um método mais intuitivo de interacção com o receptor. O segundo factor prende-se com a capacidade de dispor elementos visuais que permitem uma análise tanto do histórico do sinal como valores precisos das últimas amostras. Evita-se deste modo recorrer a ferramentas mais complexas para leitura dos ficheiros binários apenas para observar o sinal nos últimos tempos. Tudo isto torna o receptor mais simples de operar mantendo algumas das facilidades presentes num simples osciloscópio.

Por estes dois motivos consideramos que mesmo não sendo essencial é uma componente muito importante do software.

5.2.1 Bibliotecas WxPython

As bibliotecas escolhidas para implementar a interface pertencem à plataforma WxPython. Tal como o nome indica são programadas em linguagem Python. A utilização desta linguagem de alto nível vem simplificar bastante o processo de criação da interface.

Esta plataforma é baseada num paradigma de descrição. Isto é, ao invés de utilizar uma ferramenta para desenhar a interface desejada, esta tem que ser descrita através de código. Este método embora mais complexo, neste caso permite uma maior portabilidade entre sistemas operativos.

Pelo facto de wxPython ser baseado numa linguagem orientada a objectos, a dita descrição da interface faz grande recurso à utilização de classes. Estas podem variar desde um simples botão até a uma função de troca de informação entre *threads*. Todas elas estão subdivididas em diversas árvores cada uma representando um campo distinto da interface.

Algumas das classes mais importantes num projecto desta natureza são apresentados em seguida:

- `wx.Frame` : Esta classe representa um objecto que equivale a uma janela. Na sua inicialização podem ser definidos pormenores tais como título, tamanho, janela-mãe, acções possíveis sobre a janela (fechar, minimizar, maximizar, mover ...). A esta classe podem ser associados todos os tipos de objectos comuns a uma janela, tais como barras de menu, botões, imagens, texto, caixas de texto *et cetera*. Estes objectos têm as suas próprias funções de interacção e podem conter mais uma camada de variáveis associadas. No entanto não é necessária uma explicação detalhada para a compreensão da criação e gestão da GUI realizada;

- `wx.Panel` : Esta classe é derivada da `wx.Frame` tendo as mesmas capacidades. Pelo facto de se tratar de um painel ao invés de uma janela, este necessita de ser associada um objecto `wx.Frame` para que possa aparecer na interface final. A principal diferença encontra-se no facto de que esta necessita de pertencer a uma janela verdadeira pois trata-se de um painel. Sendo possível associar mais que um a uma janela, é comum serem utilizados para organizar campos dentro de uma janelas.

- `wx.lib.plot` : Trata-se de uma classe criada com o objectivo de desenhar gráficos. O seu funcionamento é semelhante ao `wx.Panel` e pode ser associado a um objecto `wx.Frame` ou `wx.Panel`, para que seja incluído na GUI.

- `wx.PyEvent` : De todas esta merece especial destaque. Visto que vamos ter pelo menos três *threads* em simultâneo, uma para a aquisição, uma para a análise e controlo e outra para a actualização da GUI, permite o envio de mensagens entre *threads*. Isto é importante pois é necessário garantir que duas *threads* não acedem simultaneamente ao mesmo recurso. No nosso caso seria a análise e a actualização da interface.

Para cada evento gerado deve ser criada a função de atendimento na *thread* destinatária. Sempre que for enviada uma mensagem, os valores apenas são alterados se não estiverem a ser utilizados. Deste modo garante-se maior robustez do *software*.

Os paradigmas usados como referência para a implementação da interface com o utilizador tal como é descrita nesta dissertação foram retirados do livro "WxPython In Action" [26].

5.2.2 *Design* e funcionalidade

Os requisitos impostos à GUI foram os seguintes:

- Deve apresentar as amplitudes do *copolar* e do *crosspolar* e respectiva fase relativa sob a forma de gráficos actualizados uma vez por segundo;
- Deve permitir alterar os parâmetros de configuração do receptor;
- Deve apresentar numericamente os valores instantâneos de todas as estatísticas relevantes;

Analisando o primeiro ponto surgem três problemas.

O primeiro tem que ver com a diferença entre a cadência com que os valores são estimados e o processo de actualização. Tal como foi dito, quando a *thread* da GUI está a actualizar os valores na interface, estes não devem ser alterados pela *thread* de sintonia ou aquisição. Por este motivo são definidos eventos temporizados que enviam uma mensagem das threads de sintonia e aquisição com os valores que querem actualizar. Sempre que a GUI estiver livre é despoletada a função de atendimento que fará a dita actualização. O temporizador está presente apenas na thread que deseja enviar a informação.

Assim, olhando o tópico 5.2.1, verifica-se que existe uma classe para este efeito.

O objecto `AcquisitionResultEvent(wx.PyEvent)` tem como única tarefa o encapsulamento de todos os valores a actualizar e o seu envio para a *thread* que gere a GUI. Por sua vez, na *thread* da GUI é feita a ligação do evento à sua função de tratamento correspondente, `UIUpdate`. Desta maneira os dados que estiverem a ser utilizados para traçar os gráficos não são alterados antes que sejam libertados, o que resolve o segundo problema: a partilha de informação entre *threads*.

Finalmente o terceiro problema é o tempo necessário para desenhar os gráficos. Tendo como objectivo a apresentação das últimas 24 horas de dados, o número de pontos pode ascender a cerca de 130 mil por gráfico. A projecção de um número tão elevado de pontos para cada um dos três gráficos pode tornar-se bastante lenta, o que significa que a capacidade de processamento está a ser desviada dos algoritmos de controlo para um simples traçar de gráficos. Isto significa que o sistema perde a agilidade que tanto se quer. Para contornar este problema, durante a primeira hora todos os pontos adquiridas são utilizadas para desenhar os gráficos. Daí em diante cada ponto é um valor retirado de entre um conjunto de valores representantes de um período de trinta segundos. Tendo em conta a variação lenta dos fenómenos de propagação, pode-se considerar que uma amostra a cada trinta segundos, no máximo, válida para se averiguar a ocorrência de algum evento de propagação.

O segundo ponto refere a acessibilidade aos parâmetros de configuração. De modo a agilizar o processo de configuração foram incluídos três submenus no menu ficheiro. A figura 5.3(a) a representa este mesmo menu. As 3 opções disponíveis são *Load Configuration*, *Edit Configuration* e *Show Configuration*. A função de cada uma corresponde exactamente ao seu nome.

A primeira abre uma típica janela para escolha de um ficheiro, onde o utilizador selecciona o ficheiro de configuração que guardou anteriormente. O resultado no SO Ubuntu com gestor de janelas GNOME é apresentado na figura 5.3(b).

Ao escolher a edição de uma configuração aparece a janela da figura 5.3(c). Aqui são apresentados todos os parâmetros possíveis de configurar. Se o utilizador quiser abrir uma configuração existente

para edição basta carregar em *Load*. Para guardar definitivamente a configuração utiliza-se a opção *Save*. Para começar a utilizar a configuração sem ter que recorrer à opção de carregamento de um ficheiro basta clicar em *OK*. Para cancelar todo o processo utiliza-se o botão *Cancel*.

Apenas três aspectos devem ser referidos em relação ao menu de carregamento e edição. A configuração que for escolhida para carregamento através da opção *Load* ou na janela de edição usando o botão *OK*, é registada através de um ficheiro *profile* que é utilizado para relembrar a última configuração usada no caso de o programa ser terminado ou haver uma falha de energia levando o computador a desligar-se. Em relação ao cancelamento do processo de edição, apenas é cancelada a alteração da configuração actual nunca anulando as alterações a qualquer ficheiro de configuração que tenha sido guardado. A última nota que se deve ter em mente é o facto destes dois submenus estarem bloqueados a partir do momento em que o *software* arranca o processo de análise e aquisição. Esta foi uma escolha tomada durante o desenvolvimento, sendo possível a sua alteração de modo a que os parâmetros possam ser alterados manualmente ao invés de automaticamente.

Finalmente, a terceira opção permite visualizar as configurações que estão a ser utilizadas bloqueando qualquer forma de as alterar acidentalmente. Esta janela pode ser visualizada na figura 5.3(d).

Por fim, o terceiro ponto está relacionado com o *design* visual da interface. É preciso ter em mente três objectivos: apresentação dos valores instantâneos das medições efectuadas; representação gráfica da amplitude do *copolar* e *crosspolar* nas últimas horas, assim como da diferença de fase entre ambos; visualização da qualidade do espectro do sinal à entrada.

A versão final da interface é apresentada na figura 5.4.

À esquerda podemos ver os três gráficos principais onde são feitos os traçados correspondentes à amplitude do *copolar* e *crosspolar* e também à diferença de fase entre os dois sinais. Estes gráficos auto ajustam-se à gama de valores recebidos mantendo sempre a melhor escala para uma correcta visualização dos dados. A informação representada pelos traçados desenhados compreende as últimas 24 horas, tendo em conta a limitação referida no primeiro ponto. Esta limitação pode ser inexistente se a máquina onde o programa é executado tiver uma maior capacidade de processamento.

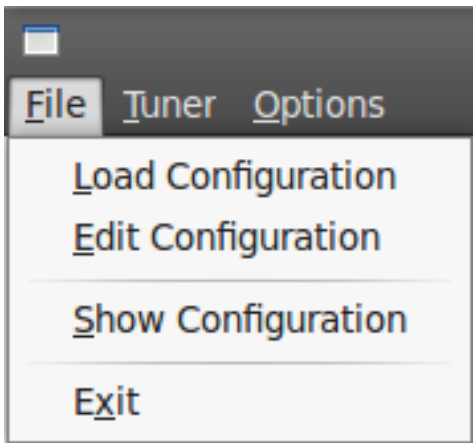
No canto superior direito encontra-se um quarto gráfico. O objectivo deste gráfico é apresentar o espectro actual do sinal. Com isto pretende-se dar ao utilizador a possibilidade de verificar a qualidade do sinal visualmente e tirar apontamentos sobre a sua forma espectral num determinado instante.

Abaixo deste gráfico encontram-se dois campos com os valores instantâneos das medições. Estes estão subdivididos em 2 grupos: os relacionados com as características do sinal e os adquiridos pela estação meteorológica. No primeiro grupo são apresentadas as seguintes medições: amplitude do *copolar* e *crosspolar*, fase relativa, frequência do sinal, CNR, NSD e variância. No segundo grupo temos: temperatura, humidade, vento e precipitação. Estes valores permitem verificar os valores actuais com maior precisão que a observação dos gráficos, assim como tornar disponíveis outras estatísticas que dispensam uma representação gráfica.

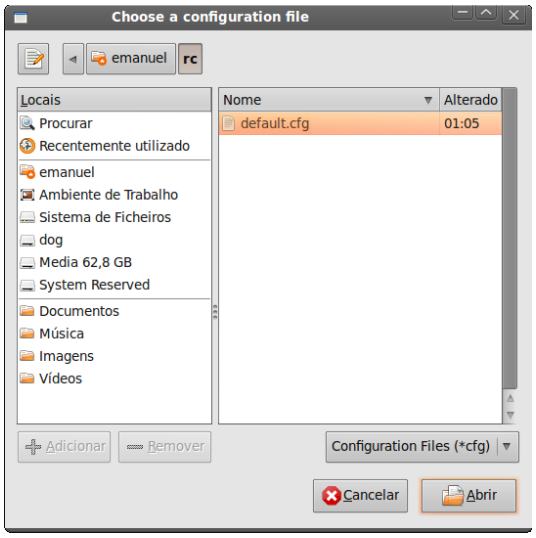
No canto esquerdo existe um botão que permite iniciar o processo de aquisição de dados. Este comuta para a função de paragem logo que o processo tome lugar. As funções deste são duplicadas no menu *Tuner*.

Adicionalmente, criou-se uma janela que surge apenas na fase de sintonização inicial. Esta pode ser vista na figura 5.5. Possui apenas dois elementos: um gráfico que apresenta o espectro do sinal *copolar* adquirido e um campo com alguns valores numéricos de estatísticas relevantes. Entre esses valores temos: *Step* e *Quality*. O valor *Step* indica quantas fases de reajuste já foram realizadas enquanto o sinal não é considerado válido, enquanto que o valor *Quality* é uma medida de qualidade que assenta no número de vezes que o sinal foi considerado aceitável antes que seja validado.

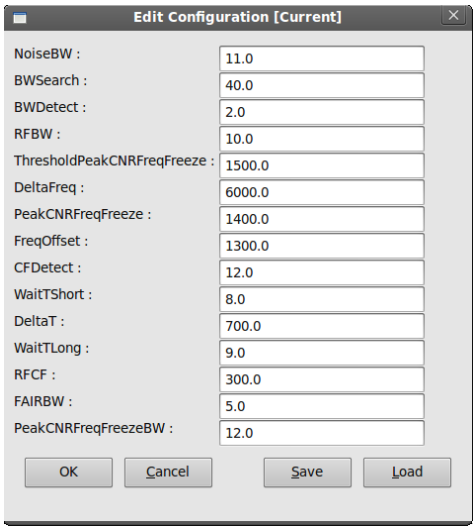
No final da sintonização esta janela é fechada automaticamente, podendo ser reaberta através do menu *Tuner*. Esta funcionalidade permite que o utilizador reveja a qualidade do sinal quando o



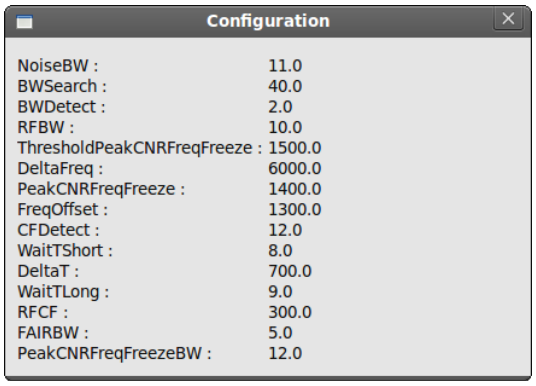
(a) Menu *File*



(b) Menu *File* - Opção *Edit Configuration*



(c) Menu *File* - Opção *Edit Configuration*



(d) Menu *File* - Opção *Show Configuration*

Figura 5.3: GUI - Opções do Menu *File*

processo teve início. Isto é especialmente útil no caso de um arranque automático.

Existem ainda mais dois pormenores da interface que merecem ser mencionados.

O primeiro é a informação fornecida na barra de estado no fundo da janela. Começando pelo lado esquerdo, temos o primeiro campo que indica o estado actual do processo. Este toma cinco valores possíveis: *Welcome*, *Tuning*, *Acquiring*, *Stopping* e *Stopped*. A mensagem *Welcome* indica que o sistema ainda não foi inicializado desde que a interface foi aberta e que está pronto para começar. A mensagem *Tuning* indica que o sistema está a sintonizar e validar o sinal para começar a aquisição, enquanto que *Acquiring* informa que a aquisição está a decorrer. Quando o utilizador dá ordem para que o programa pare, as mensagens *Stopping* e *Stopped* avisam o utilizador que o sistema está a parar ou que já parou com sucesso. O campo seguinte indica o nome da configuração que está a ser usada actualmente. O terceiro indica se a sincronização com o servidor FTP está activa. Os dois últimos campos indicam se o processo arrancou manualmente ou automaticamente e a que horas isso aconteceu. Isto permite saber se houve alguma falha de energia durante a ausência do administrador ou relembrar

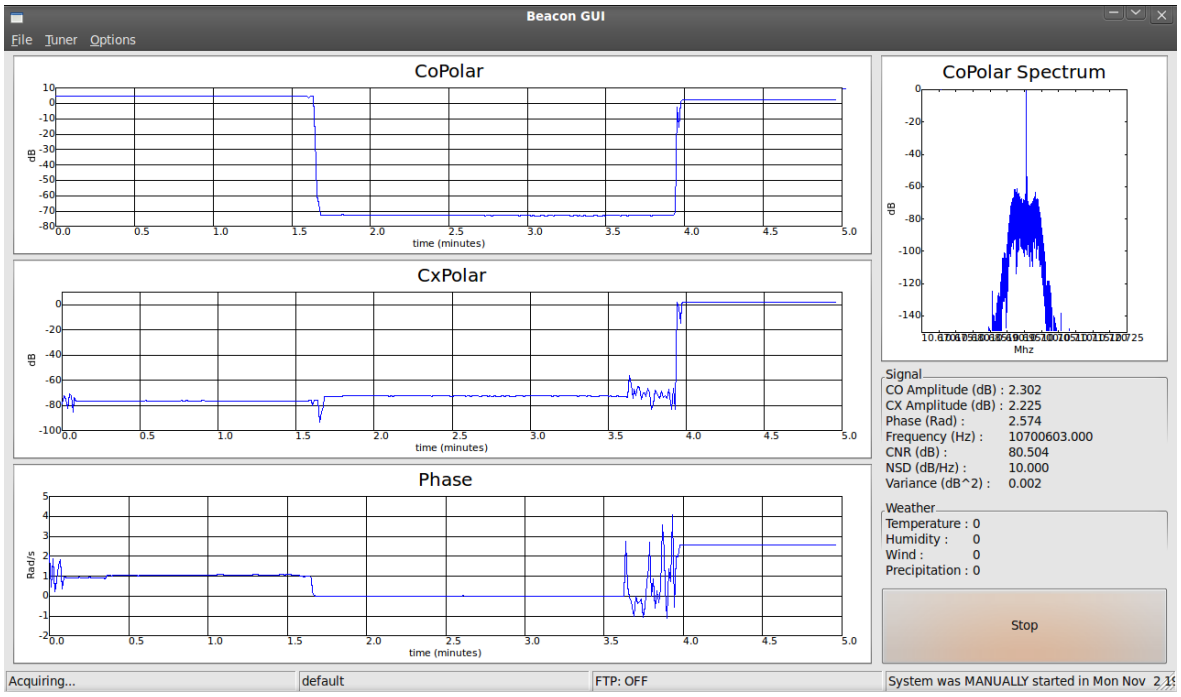


Figura 5.4: GUI final - janela de aquisição e análise

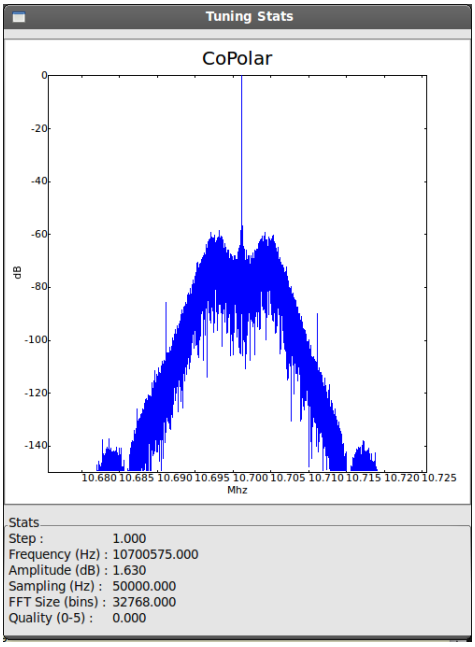


Figura 5.5: GUI final - janela de sintonização

o dia em que foi iniciado o processo corrente.

O segundo pormenor implementado é um modo de auto arranque do sistema, o que constitui uma grande vantagem garantindo que uma falha de energia não impede que o sistema fique inactivo até que seja reiniciado manualmente. Para que isto aconteça devem ser feitas duas configurações: activar na BIOS o auto arranque do PC em caso de falha eléctrica; e adicionar programa de aquisição à lista de programas de arranque do SO. Desta forma, se houver uma falha eléctrica, assim que esta seja reposta o computador ligar-se-á sozinho e ao entrar no SO o *software* corre automaticamente ao fim de um

período predefinido.

5.3 Configuração do *frontend*

Observando os diagramas da figuras 5.1 e 5.2 observamos que a configuração da PLL é o primeiro passo a dar no processo de aquisição e que esta é efectuada usando comunicação SPI.

O objectivo desta interacção é o ajuste da IF à entrada do USRP para o mais próximo possível de 10.7MHz. O motivo deste valor é o facto do filtro a cristal incluído no *frontend* estar centrado nesta frequência. Através da mistura da frequência gerada pela PLL com o sinal filtrado no segundo andar de amplificação obtemos a IF final de aproximadamente 10.7MHz. A largura da banda de passagem deste filtro é de apenas 8kHz e por isso é crucial que o ajuste seja feito com precisão e sem colocar o sinal demasiado próximo da banda de corte para garantir uma filtragem correcta do ruído e posterior digitalização do sinal.

A PLL utilizada pelo *frontend* é o modelo ADF5631[32] da Analog Devices. Após a análise do *datasheet* concluímos que é necessário programar quatro registos: N DIVIDER REG(R0), R DIVIDER REG(R1), CONTROL REG(R2) e NOISE AND SPUR REG (R3). Os campos de cada registo são apresentados na figura 5.6 retirada do *datasheet*.

N DIVIDER REG (R0)

FASTLOCK	9-BIT INTEGER VALUE (INT)									12-BIT FRACTIONAL VALUE (FRAC)												CONTROL BITS	
DB23	DB22	DB21	DB20	DB19	DB18	DB17	DB16	DB15	DB14	DB13	DB12	DB11	DB10	DB9	DB8	DB7	DB6	DB5	DB4	DB3	DB2	DB1	DB0
FL1	N9	N8	N7	N6	N5	N4	N3	N2	N1	F12	F11	F10	F9	F8	F7	F6	F5	F4	F3	F2	F1	C2 (0)	C1 (0)

R DIVIDER REG (R1)

LOAD CONTROL	MUXOUT			RESERVED	PRESCALER	4-BIT R COUNTER				12-BIT INTERPOLATOR MODULUS VALUE (MOD)												CONTROL BITS	
DB23	DB22	DB21	DB20	DB19	DB18	DB17	DB16	DB15	DB14	DB13	DB12	DB11	DB10	DB9	DB8	DB7	DB6	DB5	DB4	DB3	DB2	DB1	DB0
P3	M3	M2	M1	0	P1	R4	R3	R2	R1	M12	M11	M10	M9	M8	M7	M6	M5	M4	M3	M2	M1	C2 (0)	C1 (1)

CONTROL REG (R2)

RESYNC				REFERENCE DOUBLER	CP/2	CP CURRENT SETTING			PD POLARITY	LDP	POWER-DOWN	CP THREE-STATE	COUNTER RESET	CONTROL BITS	
DB15	DB14	DB13	DB12	DB11	DB10	DB9	DB8	DB7	DB6	DB5	DB4	DB3	DB2	DB1	DB0
S4	S3	S2	S1	U6	CP3	CP2	CP1	CP0	U5	U4	U3	U2	U1	C2 (1)	C1 (0)

NOISE AND SPUR REG (R3)

RESERVED	NOISE AND SPUR MODE				RESERVED			NOISE AND SPUR MODE	CONTROL BITS	
DB10	DB9	DB8	DB7	DB6	DB5	DB4	DB3	DB2	DB1	DB0
0	T8	T7	T6	T5	0	0	0	T1	C2 (1)	C1 (1)

Figura 5.6: Registos da PLL ADF4153

Nas notas de aplicação fornecidas pelo fabricante é aconselhada a seguinte ordem de escrita nos registos para uma correcta programação da PLL:

1. *Noise Spur Register*: com o valor `0x000003` para garantir que todos os modos de teste estão desactivados;

2. *Noise Spur Register*: com o valor `0x0003c7` para obter a melhor relação de ruído;

3. *Control Register*: com o valor `0x0003c6` configurar o modo de funcionamento e bloquear o contador;

4. *R Register*: com o valor `0x057e81`;

5. *N Register*: com o valor calculado através da equação 5.2;

6. *Control Register*: com o valor `0x0003c2` para desbloquear o contador e dar início à síntese de frequência.

Os valores atribuídos a cada um dos registos foram decididos em conjunto com o *designer* do *frontend* realizado na tese paralela.

A frequência sintetizada pela PLL é dada pela equação 5.1, onde f_{in} é a referência proveniente de um oscilador externo de alta estabilidade, nomeadamente um *ovenized crystal oscillator*, enquanto que a variável R é usada para subdividir f_{in}. Ambos os valores foram escolhidos pelo *designer* durante a realização do *frontend* com a finalidade de obter o menor ruído de fase possível. O valor MOD foi seleccionado também durante a concepção do *frontend* para garantir uma separação entre canais de 5kHz para as frequências de saída.

Resta por isso definir os valores de N_{int} e N_{frac}.

Tendo em mente que queremos deslocar um sinal centrado em 2GHz para 10.7MHz, a frequência de saída da PLL terá que ser em torno de 2.0107GHz. Este valor irá apresentar pequenas variações ao longo da execução do programa de modo a acompanhar os desvios de frequência de *beacon* ou do próprio oscilador de referência.

Após a análise da equação 5.1 chegamos à conclusão que a solução mais prática é definir um valor constante para N_{int} e ajustar a variável N_{frac} para obter as pequenas correcções desejadas. A equação 5.2 estabelece a relação entre a frequência de saída, a constante N_{int} e o valor que pretendemos para N_{frac}.

Finalmente o valor da f_{out} a sintetizar é determinado através da análise espectral, com a excepção do arranque inicial em que é predefinida para 2.0107GHz. Sabendo que a frequência de referência é de 110.9373MHz e os valores de R e MOD são respectivamente 6 e 3698, o valor de N_{int} é fixado em 108, estabelecendo uma gama de frequências entre 1.997GHz e 2.017GHz. Isto permite um seguimento flexível da frequência para deslizes mais alargados.

$$f_{out} = \frac{f_{in}}{R} \cdot \left(N_{int} + \frac{N_{frac}}{MOD} \right) \iff \tag{5.1}$$

$$N_{frac} = MOD \cdot \left(\frac{R \cdot f_{out}}{f_{in}} - N_{int} \right) \tag{5.2}$$

5.3.1 Comunicação SPI

A programação dos registos é efectuada através da interface SPI. Para isso tomou-se partido das facilidades do *hardware* USRP e *software* GNU Radio.

Por um lado a placa BasicRX utilizada possui pinos para comunicação SPI. Por outro lado existe uma implementação de comunicação SPI incluída nas bibliotecas fornecidas.

Baseado nesta função foi criada a directiva `reg_calc(F_OUT)`, onde `F_OUT` é a frequência que a PLL deve sintetizar para centrar o sinal na IF de 10.7MHz. Esta encarrega-se de calcular o valor

de N_{frac} assim como as palavras hexadecimais a escrever nos registos da PLL. A ordem da escrita dos registos é a aconselhada pelo fabricante.

Tudo isto está presente no modulo `reg_calc.py` apresentado no anexo B.

5.4 Avaliação e sintonia

A avaliação da qualidade do sinal tem por base duas componentes: potência e frequência. A estimativa de frequência e ajustes necessários apenas tomam lugar a partir do momento em que se verifica que a potência do sinal pode conduzir a resultados válidos.

Seguidamente é explicado o método de avaliação de potência para a operação de sintonia.

5.4.1 Avaliação

A estimativa de potência é feita com base no algoritmo descrito no tópico 3.3.1. Durante esta primeira fase optou-se por escolher apenas a risca de maior potência. Sendo o objectivo principal deste conjunto de testes a descoberta da frequência central, e não o valor de potência, apenas é necessário confirmar que o sinal é forte o suficiente para não ser confundido com o ruído. Para que esta aproximação seja sucedida temos que garantir que a resolução de frequência é elevada e que o limite mínimo estabelecido é superior ao nível médio de ruído. De modo a cumprir com os requisitos necessários ao método citado foram feitas as escolhas descritas nos próximos parágrafos.

A largura da banda a analisar tem um valor de 50kHz. Relembrando que a amostragem é feita a 64MHz, significa que tem que ser aplicado um factor de decimação total igual a 1280 ao sinal de entrada. A única forma de tornar isto possível é efectuando dois passos intermédios: configurar a decimação aplicada pela FPGA para o seu valor máximo e efectuar a restante redução de largura por *software*.

Pelo facto de a FPGA estar limitada a um factor de 256, a taxa de amostragem mínima transmitida por USB é de 250kHz. A este sinal é aplicado um filtro FIR com factor de decimação 5, o que se traduz numa banda de saída com 50kHz. Alguns cuidados a ter com os coeficientes escolhidos para o filtro prendem-se com a largura do filtro a cristal à saída do *frontend*. Por este motivo a frequência de corte estabelecida no filtro FIR é superior a 7kHz de modo a não rejeitar espectro com informação útil.

A partir deste ponto o sistema está apto a aplicar o algoritmo de estimação da potência do sinal. Para isso são calculadas FFTs utilizando 32768 pontos, das quais se obtém uma resolução de ≈ 1.5Hz. Esta resolução é suficientemente fina para determinar o valor da frequência central com a precisão necessária a uma boa sintonia para uma banda estreita, como é a do sinal alvo desta tese. Na prática precisa-se de apenas $0.66s$ para recolher o total de amostras, o que permite manter uma taxa de refrescamento da interface de cerca de 1Hz como o desejado.

Os valores calculados pela FFT são então usados para determinar a qualidade do sinal e verificar se é justificável um ajuste de frequência. De modo a garantir que o sinal detectado não se trata de um pico de ruído, é utilizado o valor médio das últimas cinco estimativas para análise. Esta média funciona como um filtro, evitando que picos de ruído dentro da banda de interesse tenham grande influência na classificação do sinal. Se tudo correr bem, o valor encontra-se acima do limite mínimo e é considerado seguro efectuar uma nova sintonia.

O ajuste é realizado de acordo com a frequência estimada recorrendo ao algoritmo descrito no tópico 3.3.2. Pelo facto de ser realizada uma FFT de elevada resolução em conjunto com a garantia prévia de um sinal com valor de pico estável podemos utilizar apenas uma risca para obter uma estimativa precisa. Após saber o valor, afere-se este com uma gama pré-estabelecida. Esta define

os limites para os quais o sinal é considerado estável no que se refere ao deslize de frequência. Considerou-se 3Hz para o limite de variação máximo como sendo um valor aceitável de estabilidade.

Confirmando-se que o desvio é inferior a 3Hz em relação à frequência de sintonia da placa BasicRX, não é efectuado qualquer ajuste. Caso contrário é efectuada a sintonização utilizando o valor estimado como nova IF.

Aliado ao ciclo de avaliação e sintonização encontra-se aquilo a que chamamos factor de qualidade. Na verdade este não é mais do que um contador. Este é incrementado apenas quando é feita uma avaliação em que seja confirmado que tanto a amplitude como o deslize de frequência se encontram dentro dos limites previstos. No entanto é imediatamente reposto a zero sempre que falhe em qualquer um dos testes. O processo de sintonia é dado como concluído quando este contador atingir o valor limite imposto. Esta aproximação garante que o processo de aquisição tem início apenas quando o sinal se encontra num período estável.

Em pseudo-código este processo tem o seguinte aspecto:

```
SE [amplitude > limite_mínimo] ENTÃO:

  SE [frequência_estimada > (frequência_sintonizada - 3)] ENTÃO:

    SE [frequência_estimada < (frequência_sintonizada + 3)] ENTÃO:

      factor_de_qualidade + 1
      SE [factor_de_qualidade >= 5] ENTÃO:
        ACABAR AVALIAÇÃO

    SE NÃO ENTÃO:
      frequência_sintonizada = frequência_estimada
      factor_de_qualidade = 0

  SE NÃO ENTÃO:
    frequência_sintonizada = frequência_estimada
    factor_de_qualidade = 0

SE NÃO ENTÃO:
  factor_de_qualidade = 0
```

Terminado este processo de avaliação e determinação da frequência central para sintonia da placa BasicRX, o sistema está em condições de começar a aquisição de sinal.

5.4.2 Sintonia

Este bloco tem como objectivo rastrear o espectro em busca da frequência na qual o sinal transmitido pelo TDP5 e amplificado pelo *frontend* se encontra centrado. Para melhor desempenhar esta função, adicionou-se a capacidade de reajustar a frequência de sintonização, o que se traduz num sistema com capacidade de seguimento de sinal. Isto permite garantir as melhores condições de filtragem possíveis e, consequentemente, nas melhores condições de aquisição.

Esta sintonização pode ser realizada com dois níveis distintos de precisão. O nível mais baixo consiste na alteração da frequência configurada na PLL do *frontend* para realização do processo de *downconversion*. No tópico 5.3 foi indicado que a gama de frequências sintetizáveis possui um espaçamento entre canais de 5kHz. Isto significa que o delta entre cada frequência possível é de 5kHz. Este ajuste é importante na medida em que o ruído filtrado pelo cristal depende desta frequência, e por este motivo deve-se centrar o sinal a adquirir na banda de passagem. Assim, conclui-se que o sinal deve encontrar-se na gama IF$\pm$4kHz, em que IF é a frequência de 10.7MHz à entrada do USRP e 4kHz delimita a frequência de corte do filtro a cristal.

Depois de programar o bloco gerador de frequências do *frontend* de modo a posicionar correctamente o sinal dentro da banda de passagem podemos efectuar um ajuste fino recorrendo a funções disponibilizadas pelas bibliotecas GNU Radio, as quais controlam um NCO implementado na placa mãe do USRP. O NCO tem um acumulador de fase de 32 *bits* permitindo uma precisão na ordem dos $64\text{MHz}/2^{32}$. Como pode ser inferido, existe uma enorme vantagem em utilizar esta sintonização após o primeiro passo do processo. Visto que o deslize do sinal em frequência é um processo lento, a utilização do NCO para ajustar a sintonização evita que existam saltos bruscos de frequência entre cada fase de ajuste. Se o único grau de controlo fosse ao nível da PLL estes saltos seriam na ordem dos 5kHz ao contrário de alguns Hertz.

O único caso em que a frequência da PLL deve ser novamente actualizada, é aquele em que é detectado um deslize suficientemente elevado para aproximar o sinal da banda de corte do filtro.

5.4.3 Implementação em *software*

Os ficheiros `TUNERWINmodule.py` e `tuning.py` contêm o código que implementa todo o processo descrito neste tópico. Ambos são realizados em Python, sendo que o primeiro contém a classe que constrói a janela apresentada na figura 5.5. Enquanto que o segundo contém as classes que criam e gerem o diagrama de fluxo de sinal e efectuam decisões quanto à qualidade do sinal.

Tal como foi referido anteriormente, isto poderia ser feito usando C++, no entanto os cálculos mais complexos já se encontram disponíveis nesta linguagem, o que permite utilizar apenas Python para organização do diagrama de aquisição e pós processamento dos dados.

5.4.3.1 Classe TunerFlowGraph

Começando pelo código do ficheiro `tuning.py`, as classes nele criadas são `TunerFlowGraph` e `tuner`.

A classe `TunerFlowGraph` deriva de `gr.top_block`. Entre as várias propriedades herdadas de `gr.top_block` encontram-se os métodos `run`, `wait` e `stop`, que servem para dar início, pausar ou parar o processo.

O cabeçalho da classe tem o seguinte aspecto:

```
TunerFlowGraph(CenterFrequency, FFTSize)
```

Neste é passada a frequência de sintonia do USRP e a dimensão da FFT que se deseja utilizar. Utilizando estes valores é gerado o seguinte diagrama:

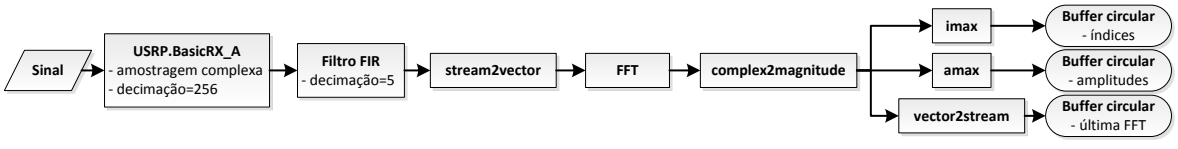


Figura 5.7: Diagrama de aquisição para avaliação e sintonia do sinal

Bloco *Sinal*:
Representa o sinal à saída do *frontend RF* com IF igual a 10.7MHz.

Bloco *USRP.BasicRX_A*:
Indica que a amostragem é realizada em modo complexo e efectua a primeira redução da largura de banda do sinal. Simultaneamente estabelece o valor inicial programado no NCO.

O módulo `usrp` da plataforma GNU Radio disponibiliza a directiva `source_c()` e `tune()`, sendo estas invocadas da seguinte forma:

```
decimation = 256
self.u = usrp.source_c(0, decimation, fpga_filename="std_2rxhb_2tx.rbf")
(...)
self.u.tune(0, self.subdev, CenterFrequency)
```

O método `source_c()` garante que a amostragem do canal 0, correspondente ao *copolar*, é complexa. Define-se que a variável `decimation` é igual a 256, valor aplicado pela FPGA ao sinal adquirido. O terceiro parâmetro escolhe o modo de funcionamento da FPGA, neste caso dois canais de recepção e dois de transmissão. Os canais de transmissão podem ser ignorados, visto que não são chamados em qualquer situação. Com isto queremos apenas garantir que temos dois canais de recepção disponíveis.

A função `tune()` define a frequência de sintonia do NCO do sistema. Esta é utilizada para converter o sinal para banda base, tornando a filtragem simples utilizando um filtro passa-baixo.

Bloco *Filtro FIR*:

Aplica a decimação restante para que a largura de banda final seja de 50kHz.

No módulo `gr` estão definidas as funções `firdes.low_pass()` e `fir_filter_ccf()`, o que possibilita a inclusão do filtro de uma forma mais simplificada através das seguintes linhas:

```
SamplingFrequency = int(64e6/decimation);
sw_decim = 5
chan_filt_decim = sw_decim
chan_filt_coeffs = gr.firdes.low_pass (1,   # gain
                    SamplingFrequency,      # sampling rate
                    1000,                   # midpoint of transition band
                    20000,                  # width of transition band
                    gr.firdes.WIN_BLACKMAN) # window type
chan_filt = gr.fir_filter_ccf (chan_filt_decim, chan_filt_coeffs))
```

Primeiro são definidos tanto a frequência de amostragem à entrada do filtro como a decimação por *software* que se deseja. Em seguida é invocada a função `firdes.low_pass()` que gera os coeficientes do filtro com as características especificadas. Neste caso opta-se por ter um ganho unitário pois o objectivo do filtro não é atenuar nem amplificar o sinal na banda de passagem. A frequência de corte é escolhida de modo a abrigar toda a banda de passagem imposta pelo filtro a cristal à saída do *frontend* RF. Após a leitura do documento produzido por Frederic J. Harris [36], seleccionou-se a janela Blackman-Harris de entre as possíveis, que segundo o autor é a variante que apresenta melhores resultados na detecção de sinusóides não afectando gravemente o desempenho da computação.

Para concluir a função `fir_filter_ccf()` gera o bloco com a decimação desejada e os coeficiente calculados anteriormente.

Bloco *stream2vector*:

O sinal transmitido através de USB pelo USRP não é nada mais que um fluxo contínuo de dados. Após o condicionamento da largura de banda é necessário efectuar o cálculo da FFT que apoia o algoritmo de análise. No módulo `gr` está presente a função `fft_vcc()` que cumpre esta tarefa, no entanto a sua entrada é um vector com o número de amostras requerido. Temos por isso que converter o fluxo para um vector. Para isso valemo-nos da função `stream_to_vector()`, do mesmo módulo, que concatena as várias amostras num único vector. Desta forma o fluxo de dados transforma-se num fluxo de vectores que correspondem à especificação do bloco seguinte.

No cabeçalho são passados o número de amostras do vector e o tamanho em *bytes* de cada uma.

Bloco *FFT*:

Após o processamento dos blocos anteriores os vectores de amostras são utilizados para obter a transformada de Fourier, sendo o resultado também ele um vector de valores complexos.

Este bloco é implementado usando a função `fft_vcc()`, à qual é passada a informação do tipo de janela a aplicar e número de pontos. Tal como no filtro foi escolhida a janela Blackman-Harris em conjunto com uma FFT de 32768 pontos.

Bloco *complex2magnitude*:

A única função deste bloco é converter todos os valores complexos do vector à saída da FFT para a sua magnitude correspondente. Este permitem detectar picos de potência mais facilmente.

Esta tarefa é executada pela função `complex_to_mag()` que se encontra no módulo `gr`, sendo o seu único parâmetro o número de elementos do vector de entrada. A saída do bloco apresenta o mesmo formato da entrada, neste caso um vector com 32678 elementos.

Blocos *imax, amax e vector2stream*:

Por um lado, o bloco *imax* devolve o índice da *bin* com a maior magnitude encontrada no vector de entrada, neste caso o pico de potência do sinal *copolar*. Por outro lado, o bloco *amax* devolve o valor dessa mesma magnitude. Utilizando ambas as informações é possível associar uma frequência ao valor de pico para posteriormente aplicar o algoritmo de avaliação e controlo de sintonia.

A função do bloco *vector2stream* é a conversão dos vectores de resultados da FFT novamente para um fluxo de dados. Este passo é necessário porque o bloco seguinte apenas aceita fluxos de dados.

Os valores à saída dos três blocos são encaminhados para *buffers* individuais de modo a guardar a história do processo até então.

Blocos *Buffer*:

Para efectuar a análise tal como foi descrita anteriormente é necessário algo que guarde o histórico dos valores estimados. Visto que não existe qualquer tipo de *buffer* com gestão de memória incluído nos módulos base do GNU Radio, surgiu a necessidade de criar o nosso primeiro bloco.

Sendo uma tarefa de manipulação de dados intensiva em tempo-real, o bloco foi criado em C++ utilizando o módulo `howto` como base de desenvolvimento. Este trata-se de um exemplo básico de como implementar um bloco em C++ na plataforma GNU Radio.

O ficheiros `howto_buffer_sink.h` e `howto_buffer_sink.cc` contêm o código que cria este bloco. Nele o vector é tratado de forma circular, isto é, assim que o tamanho máximo é atingido os valores mais antigos são eliminados e os mais recentes tomam o seu lugar. Assim é possível guardar de forma contínua o fluxo de dados sem haver problemas de excesso de consumo de memória. No entanto, o tamanho definido para o vector dita quão atrás podemos rever os valores.

O cabeçalho da classe é o seguinte:

```
howto_buffer_sink (unsigned int vector_size, float center_f,
                   unsigned int fft_size, float f_sample)
```

Neste devem ser passados o tamanho do vector, a frequência a que está sintonizada a placa BasicRX, o tamanho da FFT utilizada e a frequência de amostragem.

À parte da dimensão do vector, todos os outros valores apenas são necessários quando são chamados os métodos associados a este bloco. No âmbito desta tese foram criados os seguintes métodos:

```
howto_buffer_sink.out ()
howto_buffer_sink.out_freq ()
```

```
howto_buffer_sink.out_mag ()
howto_buffer_sink.out_fft ()
howto_buffer_sink.SetFrequency(freq)
```

Os quatro primeiros extraem os valores do histórico pela ordem correcta e efectuam a sua conversão conforme o tipo de dados guardados. O último permite alterar a frequência de sintonia com que o bloco foi iniciado.

Finalmente os três *buffers* são inicializados da seguinte forma:

```
self.buffer_freq = howto.buffer_sink(2000, CenterFrequency,
                                     FFTSize, SamplingFrequency/sw_decim)
self.buffer_amp = howto.buffer_sink(2000, CenterFrequency,
                                    FFTSize, SamplingFrequency/sw_decim)
self.buffer_fft = howto.buffer_sink(FFTSize, CenterFrequency,
                                    FFTSize, SamplingFrequency/sw_decim)
```

Isto significa que podemos aceder a um histórico de duas mil amostras tanto para valores de amplitude como dos seus índices correspondentes. O *buffer* da FFT apenas guarda a última FFT obtida, a qual é usada para actualizar o gráfico da interface.

Em suma, o diagrama explicado captura amostras complexas à entrada do USRP, guarda um histórico dos últimos valores de máximo e frequências correspondentes assim como a última FFT calculada. Estes resultados são depois utilizados pela classe `tuner` para aplicar os algoritmos de avaliação e controlo de sintonia.

O código da classe `TunerFlowGraph` pode ser visto em detalhe no anexo C.

5.4.3.2 Classe Tuner

Todo o algoritmo de avaliação e controlo de sintonia é implementado por esta classe cujo único parâmetro de entrada é a janela que recebe os dados para actualização da interface. Dentro da classe estão definidos dois métodos: `StartTuner()` e `StopTuner()`. Tal como os nomes indicam, o primeiro dá início à análise e o segundo força a sua paragem.

Durante a inicialização da classe é estabelecida a sua janela mãe e o estado é colocado em modo de execução.

Quando o método `StartTuner` é chamado, o primeiro passo consiste em criar todas as variáveis de controlo designando os seus valores iniciais. Estas são:

- Limite mínimo de amplitude – `thld`;
- Frequência central – `cf`;
- Banda estável – `cf_threshold`;
- Frequência de amostragem – `fs`;
- Número de pontos da FFT – `fsize`;
- Banda de amostragem – `bw`;
- Largura da *bin* – `bin`
- Factor de qualidade – `qf`.

Em seguida inicializa a classe `TunerFlowGraph` e dá ordem para que a aquisição de dados comece. Após o pedido espera quatro segundos pelas primeiras amostras. Ao fim deste período recolhe as últimas cinco amostras dos *buffers* de amplitude e de índices usando o método adequado, `out_mag()` para obter os valores de pico e `out_freq()` para obter as frequências correspondentes. É então calculada a média de ambos os vectores. O resultado é utilizado em seguida para verificar a qualidade do sinal e a necessidade de reajustar o NCO do USRP.

O processo de aferição do nível de sinal e a sua estabilidade é composto por uma cadeia de `ifs`, começando pela verificação do nível de sinal. Se este cumprir os requisitos, média superior a `thld`, passa para o próximo nível.

Neste ponto é testada a estabilidade de frequência do sinal. Caso o valor médio da frequência detectada se tenha afastado mais que 3Hz, em relação a `cf` em qualquer direcção o teste falha e são actualizados os valores das variáveis de controlo `cf`, `cf_threshold`, `bw` e `qf`. Esta actualização é feita usando a função `tune()` referenciada no tópico anterior. Adicionalmente, também a frequência central utilizada pelos *buffers* é alterada com recurso ao método `SetFrequency()` criado para o efeito e o factor de qualidade é forçado a zero. Caso contrário, se for confirmado que tanto a amplitude como a frequência do sinal cumprem os requisitos, a variável `qf` é incrementada uma unidade.

Ao fim de cada iteração os resultados são encapsulados num vector e enviados para a interface através do evento associado no ficheiro `events.py`.

Isto repete-se até que o factor de qualidade atinja cinco unidades, situação em que se considera que o sinal é suficientemente estável para ser iniciado o processo continuado de aquisição (5.5).

No anexo C.2 é apresentado o código detalhado desta classe.

5.4.3.3 Interacção com a interface

Durante a inicialização da interface principal é chamada a função `CreateWindow()` do módulo `TUNERWINmodule.py`, que por sua vez devolve uma variável que contém todos os elementos da janela da figura 5.5. Esta encontra-se invisível durante todo o tempo, excepto quando o processo de avaliação e sintonia decorre.

No ficheiro `events.py` está definido o evento `TunerResultEvent()` que envia à GUI os resultados da análise encapsulados. Esta atende o pedido de transferência de informação sempre que se encontrar livre. O atendimento do pedido despoleta a chamada da função `UpdateTunerWindow()` do módulo `ui.py`, que é executada na *thread* encarregada pela gestão da GUI.

O código do evento e da função gestora encontra-se no anexo D.

5.5 Aquisição, avaliação e seguimento

Esta é a função mais importante de todo o *software* do receptor, desempenhando a tarefa de seguir o sinal em frequência e simultaneamente registar todos os valores de interesse num ficheiro binário.

5.5.1 Aquisição

Ao contrário do processo de sintonia, são adquiridos dois canais, *copolar* e *crosspolar*. Esta aquisição é feita de forma continuada até que o utilizador entenda que deva ser parada ou até que ocorra algum problema com o sistema. Durante todo o período de funcionamento são calculadas em tempo-real todas as estatísticas de interesse ao funcionamento e manutenção do sistema de recepção. Entre estas encontram-se a fase relativa entre os dois canais, CNR e variância do *copolar*.

As técnicas aplicadas para estimar todos os dados registados são descritas no tópico 3.3. Tendo em mente as conclusões retiradas das simulações destes algoritmos é necessário ponderar os parâmetros escolhidos, desde a largura de banda à resolução e número de *bins* a utilizar.

No que diz respeito à escolha da largura de banda, decidiu-se manter a mesma que foi utilizada durante a sintonia, 50kHz. Esta permite conter todo espectro filtrado pelo cristal à saída do *frontend* assim como alguma da sua banda lateral, o que poderá vir a ser útil na confirmação do correcto funcionamento da filtragem e amplificação anterior ao USRP. No caso de serem detectadas anomalias, uma rápida visualização do espectro à entrada do USRP pode revelar problemas no *frontend*.

Em termos de resolução em frequência apresentamos uma abordagem diferente daquela presente no processo de sintonia. Neste caso optou-se por duas diferentes com uma relação de oito entre elas. Isto é, em qualquer instante existem duas FFTs a serem processadas, uma de elevada resolução e outra com 8 vezes menos resolução. A razão de ser desta aproximação prende-se com os fenómenos de cintilação que se fazem sentir em sinais desta natureza. Enquanto que a FFT de referência revela valores mais precisos devido ao mais longo tempo de amostragem, será insensível a variações rápidas do sinal. Estas apenas são detectáveis utilizando uma FFT com períodos mais curtos, utilizando menos amostras, que revelam as flutuações da potência do sinal ao longo do tempo. Embora em situações normais não seja esperado uma grande influência da cintilação na amplitude do sinal, o mesmo não acontece quando o clima se mostrar adverso. A vantagem está em que o nosso sistema será sensível às constantes oscilações da amplitude de sinal, e paralelamente apresenta resultados mais fiáveis recorrendo a FFTs de maior resolução. *A posteriori* estes resultados são de grande utilidade no estudo de sistemas cuja taxa de transmissão é mais elevada e por isso mais propícios a sofrer alterações provocadas pela cintilação. Pelas mesmas razões explicadas na fase de sintonia, a FFT de alta resolução tem *bins* com de aproximadamente 1.5Hz de largura, o que proporciona uma determinação da frequência central com grande precisão, essencial para um correcto seguimento de sinais de banda estreita. Como se pode deduzir, a FFT destinada a observar os fenómenos de cintilação tem *bins* de aproximadamente 12Hz, que por sua vez, apresenta resultados menos confiáveis devido à maior quantidade de potência de ruído incluída em cada uma delas.

Tal como foi referido, são adquiridos dois sinais com uma largura de 50Hz previamente filtrados numa banda de 8kHz. A amostragem de ambos é feita em modo complexo para facilitar o ajuste de frequência, isto porque faz a distinção entre frequências negativas e positivas (acima ou abaixo da frequência de sintonia). Devido à taxa de amostragem aplicada pelo USRP (64MHz) é necessário aplicar decimação para reduzir a banda para 50kHz. Um pequeno pormenor que tem que ser ajustado em relação ao que foi feito na sintonia, é o facto de a decimação feita pela FPGA em multi-canal ser metade que a presente para um único canal. Trata-se de uma limitação do USRP e em termos práticos significa que o sinal transferido pela USB apresenta 500kHz de largura ao contrário dos 250kHz no tópico 5.4.1. Tem por isso, que se aumentar o factor do filtro FIR de 5 para 10, para que sejam mantidas as condições presentes na sintonia.

Feito este condicionamento de banda inicial, reúnem-se as condições necessárias para executar os algoritmos de análise para ambas as versões da FFT: rápida e lenta. De uma maneira geral estes consistem na procura do pico de potência, cálculo da CNR, estimativa do desfasamento entre o *copolar* e o *crosspolar* assim como da amplitude do crosspolar. Adicionalmente, é registada a variância em períodos de um minuto para avaliar a estabilidade da potência do sinal.

5.5.2 Avaliação

A avaliação do sinal tem como objectivo principal obter dados precisos sobre a condição do sinal num determinado instante para que possam ser registados. Para além deste, mas não menos importante, vem a utilização das estimativas realizadas para efectuar o seguimento do sinal. Com isto queremos dizer que a frequência sintonizada não será constante, mas sim que deslizará juntamente com o sinal. Deste modo os valores medidos são mais confiáveis na medida em que é garantido que este se encontra

confortavelmente situado na banda de passagem do *frontend*.

Todos os cálculos descritos em seguida são aplicados tanto ao ramo de aquisição de alta como ao de baixa resolução.

Primeiro que tudo é necessário descobrir o valor de pico e a sua frequência respectiva. Enquanto que na sintonia temos que garantir que o sinal se encontra nas melhores condições para começar a aquisição, neste caso é permitido que sejam feitos registos e seguimento mesmo quando o sinal se encontra próximo do nível de ruído. Isto permite seguir o sinal mesmo quando existe atenuação profunda. No entanto, é introduzido um novo grau de complexidade, visto que segundo as simulações a diminuição da CNR torna as estimativas menos precisas. Por esta razão justifica-se a utilização de uma média pesada para melhorar a qualidade da determinação da frequência de pico. Por um lado, o método requer que seja definido uma largura de banda lateral, a qual é utilizada para auxiliar os cálculos. Por outro lado, sabemos através de simulações que um menor número de riscas, equivalente a uma banda mais reduzida, melhora a precisão quando o sinal se encontra numa situação extrema. O que nos leva a tornar este parâmetro variável em função da CNR. Isto é, dependendo da CNR actual o receptor vai utilizar uma largura de banda mais adequada para avaliar o sinal.

Sabendo a frequência central e a amplitude do *copolar* podemos extrair a amplitude do *crosspolar* e a sua fase relativa. O sinal *cx* é correlacionado com o sinal *co* calcula-se a sua amplitude de acordo com o algoritmo descrito no tópico 3.3.4. Olhando novamente às simulações realizadas para a estimativa da fase relativa, conclui-se que a largura de banda usada neste algoritmo não tem qualquer influência na sua precisão. Esta é a razão pela qual não se define nenhum valor específico e utiliza-se a mesma largura escolhida para o *copolar*. Após aplicar o algoritmo descrito em 3.3.3 estão determinados os valores principais a registar: amplitude do *copolar* e do *crosspolar* e a sua fase relativa.

Neste pontos estamos aptos a computar o valor da CNR, mas para isso é preciso primeiro estimar o NSD. Escolheu-se utilizar uma gama de 4kHz em torno da banda do sinal para estimar o valor médio do ruído por Hz. Sabendo que o sinal tem um espalhamento na ordem dos 50Hz, significa que na prática o espectro de ruído capturado está nos intervalos $[f_c + 50, f_c + 2000]$Hz e $[f_c - 2000, f_c - 50]$Hz. É de notar que se deixa uma pequena margem entre o limite da banda do sinal e a banda de ruído. Esta permite ter a certeza de que não se está a considerar uma zona onde a potência do *copolar* ainda pode ser relevante em relação ao nível de ruído.

Feita a média da potência do ruído por Hertz resta apenas usar a equação 3.15. Este valor será utilizado no próximo ciclo de avaliação para determinar o número de *bins* a escolher. Adicionalmente será utilizado no corrente ciclo para verificar se o sinal respeita as condições necessárias para efectuar o seguimento.

O último valor a ser obtido trata-se da variância de potência do sinal *copolar*. Esta é calculada em períodos de um minuto e reposta a zero ao fim desse tempo. Através deste valor podemos verificar com mais precisão a estabilidade sem necessitar de observar os traçados do gráfico.

De novo, o algoritmo aplicado é explicado em detalhe no tópico 3.3.6.

Após fazer todas as estimativas, os dados recolhidos são armazenados num ficheiro binário. Foram tomados dois cuidados na geração destes registos. Em primeiro lugar, os dados são guardados em tempo real, o que evita a perda de dados em caso de falha. Se ao contrário deste método fossem escritos os ficheiros apenas ao final de cada período de captura, haveria o risco de ocorrer algum problema que bloqueasse o sistema e todos os dados até então estariam perdidos. Em segundo lugar, é feito um cabeçalho com a informação necessária para recuperar todos os dados e localizá-los temporalmente. Neste cabeçalho constam os valores da hora de início, frequência de amostragem, número de pontos

da FFT e número de amostras. Conhecendo a estrutura deste cabeçalho facilmente se criou o *script* MATLAB que efectua a leitura dos dados.

Com isto concluí-se a descrição conceptual da tarefa de análise e registo.

5.5.3 Seguimento

No que diz respeito ao seguimento do sinal existem três factores a ter em conta: o natural deslizamento de frequência ao longo do tempo; a garantia de que o sinal se encontra dentro dos limites do último filtro da cadeia de amplificação do *frontend*; e, finalmente, o desaparecimento do sinal.

Começando pelo deslizamento natural, não se espera variações diárias superiores a 4kHz. No entanto devido à estreita largura tanto do sinal como do filtro a cristal, esta variação relativamente pequena poderá conduzir à adulteração das medições mesmo em situações de recepção perfeitas. Tal como foi dito, é aplicado um filtro FIR ao sinal para redução da largura de banda. À medida que o sinal se distancia da frequência de sintonia aproxima-se da banda de corte do filtro. Esta movimentação pode traduzir-se numa aparente perca de potência, quando na verdade é devido à função de transferência do filtro.

Para contrariar este fenómeno, a cada iteração é considerada a opção de alterar o valor do NCO. Usando o valor da CNR como referência de qualidade, é estabelecido um limite mínimo para validar a mudança de frequência. Visto que o deslize é um fenómeno lento, é possível usar com segurança as estimativas dos últimos segundos para melhorar o seguimento. Ao invés de usar apenas um valor de CNR, é efectuada a média das últimas cinco medições para filtrar picos de ruído ou percas instantâneas de sinal. Após assegurar que a CNR está acima do mínimo verificamos se houve um deslize que justifique uma re-sintonização. Novamente recorre-se à média dos últimos cinco elementos, desta vez de frequência, sendo que indicam a frequência central dominante nos instantes precedentes. No que diz respeito a este ajuste fino, considera-se significativa qualquer variação superior a 3Hz. Desta maneira evita-se que mais tarde seja feito um salto de frequência maior, o que prejudica a aquisição do sinal no breve período que lhe sucede durante o qual o sinal se encontra em fase de estabilização.

No entanto é necessário ponderar um caso mais grave, um deslize de frequência a longo prazo que coloca o sinal perto da banda de corte do filtro do *frontend*.

Empiricamente, verificou-se na experiência ligada ao trabalho indicado em [31], durante o ano a frequência varia alguns kHz, o que justifica a alteração da IF de entrada no sistema de aquisição. Devido à capacidade de comunicação implementada entre o USRP e o sintetizador de frequência montado no *frontend*, existe a capacidade de alterar o oscilador local de modo a converter a sua frequência para os 10.7MHz finais. Pelo facto do espaçamento mínimo entre frequências adjacentes ser 5kHz e a largura do filtro situar-se perto dos 7kHz, definiu-se que um deslizamento a longo prazo superior a 3kHz justifica uma alteração ao nível do *hardware* de amplificação e filtragem. Este ponto encontra-se na intercepção virtual entre os dois filtros possíveis, um centrado na IF actual e um segundo centrado na IF adjacente. Isto significa já existe algum efeito de modulação em amplitude antes da correcção ser efectuada. Trata-se de um problema ao qual não podemos fugir, pois prende-se com uma limitação imposta pela PLL do *frontend* no que toca a espaçamento entre canal. No entanto considerou-se este método uma alternativa válida na mitigação desta restrição.

Após fazer o ajuste da frequência sintetizada pela PLL é necessário reajustar o NCO pois fica com um *offset* de cerca de 2k, dado pelo ajuste de 5k menos o desvio de 3k ocorrido. No final destes dois passos está integrado o algoritmo de ajuste de deslizes de longo prazo. Embora sejam feitos com menos regularidade, são um passo em frente no que diz respeito à manutenção do sistema. Sendo feito de forma automática, evita que o utilizador tenha que fazer periodicamente ajustes manuais do

oscilador de referência principal do receptor.

O terceiro merece especial atenção. Quando o sinal diminui de amplitude tornando-se indistinguível entre o ruído o sistema entra em modo de espera. Isto é, o ajuste de frequência é estagnado enquanto não existem provas de que o sinal recuperou.

A estagnação da frequência está coberta pela política de CNR mínima. No entanto não podemos considerar o primeiro indício de recuperação do sinal como válido. A amplitude do sinal não decresce instantaneamente e o mesmo é verdade para a sua subida. O que significa que quando começar a recuperar existe um período em que o nível deste é da mesma ordem que o nível mínimo estabelecido para a CNR. Por este motivo estabelece-se que a partir do momento que se detecta novamente o sinal, se aplica um tempo de espera de um minuto para garantir que se encontra estável. Apenas após este período se volta a ponderar o ajuste da frequência. Assim o sistema corre menos riscos que julgar uma qualquer perturbação esporádica como sinal principal, ou mesmo de alterar a IF numa altura em que o sinal ainda não se encontra totalmente estável.

Dado o facto que as condições que provocam uma atenuação elevada na região onde decorre a experiência, Aveiro, não prevalecem durante mais que alguns minutos, podemos admitir que o sinal se encontra posicionado sensivelmente no mesmo ponto do espectro que no momento em que se tornou indetectável.

Os três pontos discutidos acima representam todos os cuidados tomados pelo sistema ao nível do seguimento de sinal. Em suma, é verificado se o sinal é detectável. Se o for e não se encontrar no período de espera confronta-se o nível da CNR com o mínimo. Se tudo estiver em ordem averigua-se a necessidade de um ajuste do *frontend*. Caso seja necessário, incrementa-se ou decrementa-se a frequência sintetizada da PLL juntamente com a frequência das placas BasicRX. Caso não seja e se for justificado, apenas se ajustam as placas do USRP.

5.5.4 Implementação em *software*

Os ficheiro que contém toda a implementação da aquisição, avaliação e seguimento de sinal são: `acquisition.py`, `howto_co_cx_stat.h/.cc` e `phase.h/.cc`. Como suporte existe o módulo `reg_calc.py` que efectua a programação da PLL. No primeiro dos ficheiros mencionados estão definidas duas classes da maior importância neste processo: `AcquisitionFlowGraph()` e `acquire()`. Em seguida são apresentados detalhes sobre cada uma.

5.5.4.1 Classe AcquisitionFlowGraph

À semelhança da classe `TunerFlowGraph()` do processo de sintonização, esta tem como objectivo definir o diagrama de aquisição descrito em 5.5.1. Como parâmetros de entrada são passados os nomes dos ficheiros binários utilizados para registar os dados, a frequência sintonizada e o tamanho da FFT. Como resultado é gerado o diagrama da figura 5.8.

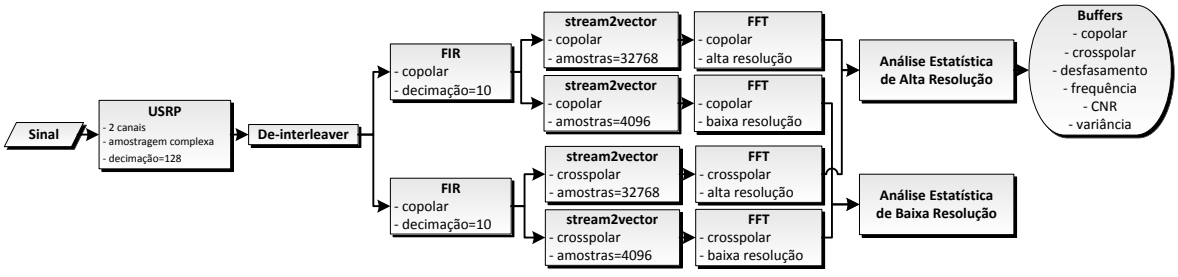


Figura 5.8: Diagrama de aquisição para os registos binários

Blocos Sinal e USRP: Representam os mesmos elementos que os descritos em 5.4.3.1. As únicas diferenças a salientar são por um lado o facto de agora o sinal *crosspolar* também ser considerado à entrada do USRP; por outro lado o factor de decimação é 128 pelas razões descritas em 5.5.1.

Bloco De-interleaver: Visto que a comunicação USB é do tipo série, os dados vêm multiplexados no tempo e por isso apresentam-se numa trama igual à da figura 4.6. O que este bloco faz é recuperar os fluxos de dados de cada canal tal como estas foram adquiridas. O resultado são duas streams de saída, uma para o copolar e outra para o crosspolar a um ritmo de 500k amostras por segundo.

Para criar o bloco apenas é necessário saber o tamanho em bytes de cada amostra.

```
gr.deinterleave(gr.sizeof_gr_complex)
```

Blocos FIR: Mais uma vez, tal como na função de sintonia, têm a função de aplicar um filtro passa-baixo ao sinal ao mesmo tempo que efectuam a decimação restante de modo a condicionar a largura de banda para os 50kHz como é desejado. Os coeficientes do filtro são gerados da mesma forma que na classe `TunerFlowGraph`.

Neste caso são feitos dois filtros pois são adquiridos dois sinais distintos.

Blocos stream2vector: A diferenciação que se falou entre aquisição rápida e lenta, para uma melhor apreciação dos fenómenos de cintilação, é feita neste ponto. Basicamente, geramos dois fluxos de vectores com diferentes tamanhos para cada um dos sinais. Ao gerar vectores com tamanhos diferentes surge a oportunidade de realizar uma FFT de alta resolução e outra de baixa resolução. Como é fácil de concluir aquela que requer menos amostras apresenta um ritmo de cálculo de resultados mais elevado, e por isso ideal para observar a cintilação.

Blocos FFT: Também aqui não existem surpresas, sendo que a única novidade é o facto de existirem dois blocos de duas FFT, uma de 32768 e outra de 4096 amostras. A primeira apresenta uma resolução de aproximadamente 1.5Hz enquanto que a segunda 12Hz.

Blocos de Análise Estatística: Tal como o nome indica estão encarregados de calcular todos os valores descritos em 5.5.2. Pelo facto de serem funções específicas do receptor desenvolvido para esta tese, ambos foram criados em C++ e baseados no exemplo `howto` referido no tópico 4.4.2.2. Para além dos cálculos, adicionam os resultados de cada iteração aos registos binários de acordo com as regras descritas anteriormente. Como parâmetros de configuração temos a frequência central, taxa de amostragem, número de pontos da FFT e o nome do ficheiro de registo. Assim existe a capacidade de diferenciar o bloco de alta resolução do de baixa, eliminando a necessidade de criar dois blocos.

Em seguida é escrito o cabeçalho do registo com o seguinte aspecto:

```
typedef struct
{
  unsigned long Rows;
  unsigned long Columns;
  unsigned fft;
  unsigned sampling;
  char Date[25];
} MATLAB_MATRIX_HEADER;
```

Juntamente com esta acção é iniciado o contador que marca o período de cálculo da variância assim como o valor por defeito da CNR para a primeira iteração.

O primeiro passo da análise é realizar o *shift* da FFT. O vector resultante do bloco FFT não está ordenado da frequência mais baixa para a maior. Na verdade a primeira metade corresponde à gama $[F_a, F_a + F/2]$ e a segunda à gama $[F_a - F_a/2, F_a]$.

Feito isto, podemos começar a procurar o pico de potência no vector correspondente ao *copolar*. Como neste diagrama não existe um bloco de conversão para magnitude a seguir à FFT é necessário o cuidado de olhar ao módulo ao invés do valor complexo do vector de entrada.

Após determinada a *bin* de máxima potência temos que decidir o número de riscas a utilizar. As riscas são determinadas em função da largura de banda. Desta maneira o algoritmo de decisão torna-se mais genérico baseando-se na resolução, banda e CNR. O limite máximo imposto é de 50Hz pois trata-se da largura de banda do sinal. Já o mínimo é de 12Hz que tal como foi observado nas simulações, cujos resultados estão ilustrados nas figuras 3.7 e 3.9, para uma CNR inferior a 40dB/Hz uma banda de 12Hz conduz a melhores resultados.

O código seguinte implementa esta escolha de uma maneira simples:

```
float band;
if(cnr < 40)
{
  band = 12;
}
else
{
  band = 50;
}
resolution = d_sampling/float(d_fftsize);
band_bins = ceil(band/resolution);
```

Estabelecida a banda a utilizar são criados dois vectores auxiliares que contêm os elementos de interesse tanto do *copolar* como do *crosspolar*. Ao mesmo tempo que isto é feito aproveitamos para calcular a potência do sinal *co*, o que poupa um ciclo de iterações. As linhas seguintes demonstram este ciclo:

```
float signal_pow = 0;
unsigned nbins_co = 0;
vector<gr_complex> band100_CO;
vector<gr_complex> band100_CX;
for (unsigned int i = max_index-band_bins/2;
                  i <= max_index+band_bins/2;
                  i++, nbins_co++)
{
  signal_pow += abs(co_fft_ord.at(i)) *  abs(co_fft_ord.at(i));
  band100_CO.push_back(co_fft_ord.at(i));
  band100_CX.push_back(cx_fft_ord.at(i));
}
```

Tendo o valor de potência do sinal *co* podemos calcular a CNR. Para isso executa-se um ciclo semelhante ao anterior para obter o vector auxiliar com o ruído correspondente a 4kHz de banda e no final calcula-se a sua densidade espectral. A cópia de valores contempla uma margem de 200Hz entre o limite da banda de sinal e o primeiro elemento de ruído adicionado. Desta forma garante-se que a influência do sinal neste cálculo é mínima.

```
float noise_pow = 0;
unsigned nbins_noise = 0;
unsigned int b200 = ceil(200/resolution);
unsigned b2000 = ceil(2000/resolution);
vector<gr_complex> band4000_Noise;
```

```
/* abaixo do sinal */
for(unsigned int i = max_index-b200-b2000;
                 i < max_index-b200;
                 i++, nbins_noise++)
{
  band4000_Noise.push_back(co_fft_ord.at(i));
  noise_pow += abs(co_fft_ord.at(i)) * abs(co_fft_ord.at(i));
}

/* acima do sinal */
for(unsigned int i = max_index+b200;
                 i <= max_index+b200+b2000;
                 i++, nbins_noise++)
{
  band4000_Noise.push_back(co_fft_ord.at(i));
  noise_pow += abs(co_fft_ord.at(i)) *  abs(co_fft_ord.at(i));
}

float noise_pow_hz = noise_pow/(nbins_noise*resolution);
cnr = 10*log10(signal_pow/noise_pow_hz);
```

Para o cálculo de variância é aplicado o algoritmo descrito em 3.3.6.

Finalmente é feita a estimativa de fase relativa e amplitude do sinal *crosspolar* segundo os princípios referidos em 3.3.3 e 3.3.4. O algoritmo encontra-se implementado nos ficheiros `phase.h/.cc`. Para utilizá-lo basta chamar a função `phaseEstimation()` cujo o cabeçalho é o seguinte:

```
int phaseEstimation(std::vector<gr_complex> VectorCx,
                    std::vector<gr_complex> VectorCo,
                    float *CxPhase,
                    float *CxAmp,
                    float *CoAmp);
```

No final de todos os cálculos os ficheiros são escritos no registo binário cujo cabeçalho é actualizado com o novo número de amostras e dispostos à saída do bloco.

Blocos Buffer: Cada um dos valores descritos no bloco tem o seu próprio *buffer*, sendo que cada um destes é igual ao utilizado no diagrama da figura 5.7 e descrito em 5.4.3.1. Os valores aqui contidos fornecem dados suficientes para o controlo do receptor. Para todos eles é estabelecida uma capacidade capaz de acomodar dados suficientes para representar as últimas 24 horas.

5.5.4.2 Classe Acquire

Esta classe tem como objectivo interpretar os dados guardados pelo diagrama de fluxo da figura 5.8 nos *buffers*, e assim aplicar as ordens de controlo do USRP e da programação da PLL do *frontend*.

Para inicializar esta classe apenas é necessário indicar qual é a janela mãe, à qual serão enviados os dados da aquisição. Também durante o processo de inicialização são configuradas todas as variáveis de controlo com os seus valores por defeito.

Para a avaliação da qualidade e sintonia são usados os mesmos parâmetros que no processo descrito em 5.4.3.2. Adicionalmente encontra-se a última frequência programada na PLL para sejam detectados os desvios que aproximam o sinal da banda de corte do filtro a cristal.

Outras variáveis que também estão presentes apenas nesta fase são os contadores que verificam a duração do registo, se houve mudança de dia e há quanto tempo o sinal se encontra estável.

As funções que compõe a classe são `StartAcquire()`, `StopAcquire()`, `NewLog()`, `CheckEndOfDay()` e `ReturnValues()`.

A função `StopAcquire()`, à semelhança da `StopTuner()`, tem como único objectivo quebrar o ciclo ininterrupto de acquisição/avalição/sintonização.

O cerne de todo o processo é o método `StartAcquire()`, o qual é descrito na sua totalidade nos próximos parágrafos.

Como seria de esperar a sequência de passos que vai desde a aquisição até à sintonia é muito semelhante ao que foi feito antes. Em primeiro lugar dá-se início ao diagrama de aquisição que corre paralelamente à avaliação. Depois com base nos dados que vão preenchendo os *buffers* são feitas médias com as quais se verifica a necessidade de sintonia. E é aqui que esta fase diverge da sintonia inicial.

Enquanto que nessa fase a preocupação era de obter uma boa amplitude que nos garantisse que o sinal era o correcto, aqui vamos um pouco mais além. Ao invés da amplitude é utilizada a CNR. Isto significa que podemos seguir o sinal com mais confiança pois sabemos exactamente quão próximo este se encontra do ruído. Para além deste aspecto, devemos relembrar que foram aplicadas técnicas mais sofisticadas de estimação da potência do sinal, e logo por esse facto a detecção é mais rigorosa. Para além da precisão das medições encontra-se o ajuste do sinal na banda de passagem do filtro do *frontend* externo ao USRP.

Recordando o processo de verificação da fase anterior temos esta sequência, que descreve as condições necessárias para que se justifique um ajuste de frequência:

1. Se a potência for superior ao limite mínimo então:
2. Verifica-se a estabilidade da frequência. Se houve deslocamento então:
3. Altera-se a frequência do NCO.

O que acontece nesta fase é a introdução de uma verificação extra entre o ponto 2 e o 3. Esta consiste em confrontar a frequência actual com a frequência central do filtro a cristal.

O resultado desta alteração resulta no seguinte pseudo-código:

```
SE [estável] ENTÃO:

  SE [frequência_estimada > frequência_sintonizada_BasicRX + 3] OU
  SE [frequência_estimada < frequência_sintonizada_BasicRX - 3] ENTÃO:

    SE [frequência estimada > frequência_sintonizada_PLL + 3000] ENTÃO:
      AJUSTA A FREQUÊNCIA DA PLL (ANTERIOR + 5000)
      AJUSTA A FREQUÊNCIA DA IF (PLL - 2000)
    SE [frequência estimada < frequência_sintonizada_PLL - 3000] ENTÃO:
      AJUSTA A FREQUÊNCIA DA PLL (ANTIGA - 5000)
      AJUSTA A FREQUÊNCIA DA IF (PLL + 2000)
    SE NÃO ENTÃO:
      AJUSTA A FREQUÊNCIA DA IF (frequência_estimada)
    ALTERA FREQUÊNCIA DE CONFIGURAÇÃO DOS BUFFERS E BLOCOS DE ANÁLISE

SE NÃO ENTÃO:
  ESTÁ ESTAGNADA A SINTONIA
```

A estagnação não é mais que um contador que verifica o tempo decorrido desde que o sistema alterou o seu estado de não estável para estável. A sua implementação pode ser tão simples quanto:

```
SE [CNR > limite_minimo] ENTÃO:
```

```
    SE [contador está parado] ENTÃO:
      INICIA O CONTADOR
    SE [contador > 1 minuto] ENTÃO:
      estável = verdade

  SE NÃO:
    CONTADOR BLOQUEADO
```

Tudo isto é repetido uma vez por segundo.

No fim de cada ciclo é chamada a função `ReturnValues()`. Esta verifica o tempo decorrido desde o início da aquisição para que possa ajustar o número de amostras a utilizar para traçar os gráficos da GUI. Tal como foi antes dito, durante a primeira hora é utilizada a totalidade dos dados. Ao fim deste período é utilizada apenas uma amostra em cada trinta, o que equivale a aproximadamente um ponto por cada vinte segundos. Voltamos a relembrar que devido à taxa de variação do sinal este é um método válido a longo prazo, reduzindo a carga de processamento atribuída a tarefas secundárias como é o caso do refrescamento da interface.

Após a escolha das amostras para os três gráficos, estas são concatenadas em três vectores e empacotadas junto com o resto dos resultados para serem enviadas para a *thread* principal que gere a interface.

Em todos os casos as amostras seleccionadas pertencem aos *buffers* de alta resolução.

5.5.4.3 Funções Auxiliares

Tal como foi mencionado antes existem algumas funções que auxiliam o processo de registo: `CheckEndOfDay()`, `treeGen()` e `NewLog()`.

`CheckEndOfDay()`: No início da aquisição é iniciado um temporizador que expira uma vez por segundo. Sempre que isto acontece esta função é executada, na qual é então verificado se ocorreu mudança de dia. No caso de ser verdade é lançado o processo de criação do novo registo. Para isso são chamadas as funções `treeGen()` e `NewLog()`.

`treeGen()`: Esta função encontra-se no ficheiro `TREEGENmodule.py` criado especialmente para este caso. A sua função é criar a árvore de pastas e devolver o caminho completo incluindo o nome do ficheiro a criar para os registos de aquisição rápida e lenta. A árvore é criada no local que o utilizador definir e têm os seguintes níveis:

1. `->PASTA ANO`
2. `-->PASTA MÊS`
3. `--->PASTA RÁPIDO/LENTO`

Sendo que no final o caminho terá o seguinte aspecto:

```
/user_selected_folder/2009/Nov/HiRes/2009Nov7_0
/user_selected_folder/2009/Nov/LowRes/2009Nov7_0
```

O ficheiros mantêm o nome para um dado dia alterando apenas o algarismo sufixado, indicando a ordem pela qual foram criados nesse mesmo dia.

`NewLog()`: Este processo pode ser lançado de duas formas. A primeira foi mencionado no parágrafo anterior e corresponde a um final de dia. A segunda maneira é conclusão de 6 horas de registo contínuas.

Em qualquer dos casos a seguinte sequência de passos é realizada:

1. Chamada da função `treeGen()`;
2. Troca do nome do registo utilizado pelos blocos de análise estatística utilizando o método `create_new_log(filename)`, o qual permite alterar o ficheiro de escrita durante o processo de aquisição sem que sejam corrompidos os registos;
3. Reinicialização do temporizador que marca o final do período máximo de 6 horas.

No final deste conjunto de operações o registo de ficheiro está a ser efectuado no novo destino criado em função da data actual.

`Upload()`: Como extra encontra-se uma opção em estado experimental de sincronização da árvore de ficheiro para um servidor FTP, implementada no ficheiro `FTPmodule.py`. Esta sincronização ocorre apenas no sentido do FTP, verificando os ficheiros que não se encontram disponíveis remotamente e transferindo-os. No final é criado um registo contendo o caminho dos ficheiros e a informação acerca do sucesso do *upload*. Todos os ficheiros que não forem enviados correctamente ficam em lista de espera para uma próxima tentativa.

5.5.4.4 Interacção com a interface

Para que não existam conflitos no acesso à memória por parte da *thread* de aquisição e da GUI, foi criado o evento `AcquisitionResultEvent()` presente no módulo `events.py` que pode ser visto no anexo D.

Na *thread* principal encontra-se a função de atendimento `UIUpdate()`. Dentro da função são desempacotados os dados referentes aos traçados dos gráficos de potência dos sinais *copolar* e *crosspolar*, fase relativa e valores instantâneos de todas as estimativas efectuadas. Em seguida são actualizados todos os campos de valores numéricos e redesenhados os gráficos.

Devemos reiterar que a utilização de eventos é essencial para a construção de uma interface robusta para gestão de tarefas paralelas, como é o caso do *software* do receptor descrito nesta dissertação.

Capítulo 6

Testes e Discussão de Resultados

Nesta capítulo são apresentados os vários testes efectuados ao sistema.

6.1 Linearidade

Um dos objectivos principais de um sistema de medição é a sua linearidade. Por este motivo o primeiro teste apresentado diz respeito à linearidade do sistema.

Com o fim de averiguar a linearidade do sistema completo foi feita a seguinte montagem:

- Conectou-se um gerador HP8671B ao segundo andar do *frontend*, correspondente ao hardware implementado em [28];
- O gerador injecta um sinal a 2GHz fazendo variar a potência entre -115dBm e -80dBm;
- A saída do *frontend* foi conectada ao canal A do USRP, tal como mostra na figura 5.1;
- Deu-se início ao processo de aquisição e durante a fase descrita em 5.5 são retirados os valores estimados de potência à saída do *frontend*.

No final a configuração pode ser vista na figura 6.1, em que é essencialmente o diagrama da figura 5.1 em que o andar do LNA é substituído pelo gerador da HP que simula a IF de 2GHz esperada.

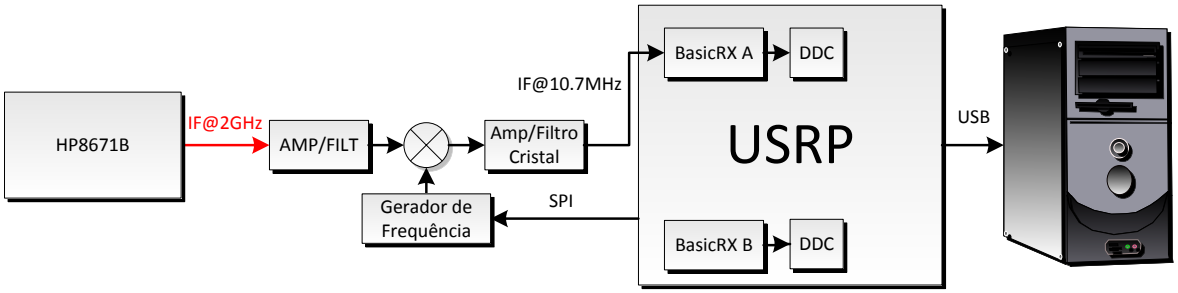
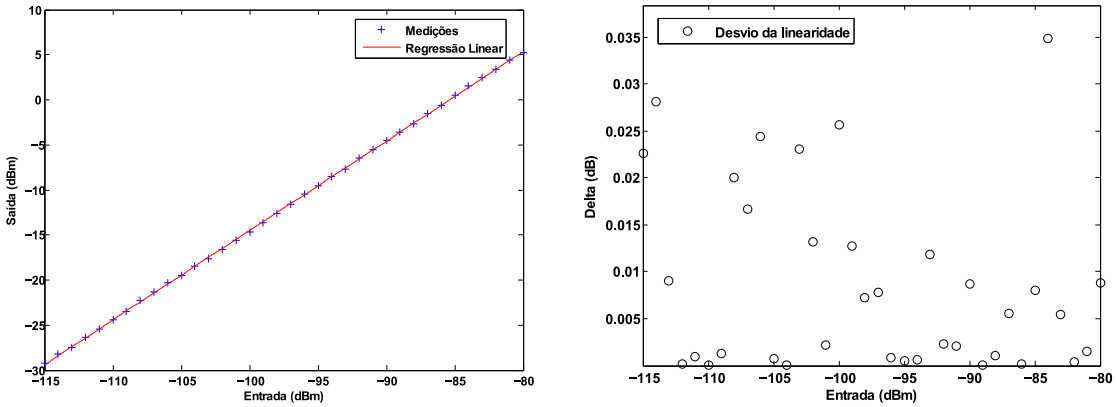


Figura 6.1: Diagrama da montagem do teste

Utilizando os valores retirados foram feitos os gráficos da figura 6.2. Neste são apresentados os pontos estimados durante os testes, assim como a recta com os parâmetros da regressão linear:

$$r^2 = 0.9999$$
$$m = 0.9927(dB)$$
$$b = 84.7748(dBm)$$
$$y = 0.9927x + 84.7748(dBm)$$

Como é possível observar os valores medidos estão distribuídos de forma muito próxima da recta de linearização. Este facto em conjunto com o valor de r^2, muito próximo da unidade, indicam que os resultados obtidos são muito lineares para a gama de valores de entrada esperadas. Podemos por isso concluir que esta componente do *software* de instrumentação tem um comportamento altamente linear como era desejado.



(a) Linearidade do sistema completo

(b) Desvio à recta de linearização

Figura 6.2: Resultados da medição do copolar

6.2 Estimativa de potência do sinal *crosspolar*

Este teste teve como objectivo verificar a linearidade e o intervalo de confiança com que a potência do sinal *crosspolar* pode ser estimada.

De modo a simular as condições reais foi utilizado um circuito somador para adicionar um nível de ruído igual a -40dBm à entrada do USRP. Para que o *crosspolar* seja representado de forma mais realista, este foi derivado do *copolar* recorrendo a um *power splitter* e atenuando-o 20dB. Assim, o sinal *cx* acompanha as variações do *copolar* mantendo constante o desvio de fase proveniente da sua passagem pelo *power splitter*.

Com base nesta montagem foram retirados os valores dos quais resulta o gráfico da figura 6.3. Estes representam a estimativa do sinal *cx* para uma variação da potência do *copolar* entre -30dBm e -5dBm.

Como se pode observar o valor estimado para o *crosspolar* acompanha o *copolar* à medida que este perde potência. No entanto, a partir do momento em que a potência do sinal *cx* atinge o limiar do nível de ruído esta deixa de ser confiável. Por este motivo a recta de linearização apresentada tem em conta apenas os sete pontos do lado direito do gráfico. Utilizando apenas este conjunto obtemos os seguintes parâmetros de linearização:

$$\begin{aligned} r^2 &= 0.9897 \\ m &= 1.0346(dB) \\ b &= 84.2229(dBm) \\ y &= 1.0346x - 84.2229(dBm) \end{aligned}$$

Embora se possa verificar que não apresenta uma linearidade tão elevada como a estimativa do *copolar*, pode-se confirmar que é detectado e que a sua amplitude estimada é de forma correcta nos casos em se encontra acima do nível de ruído.

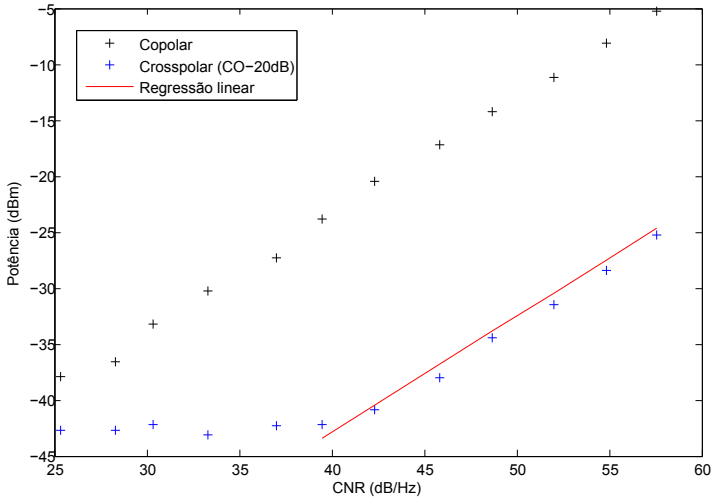


Figura 6.3: Teste de linearidade da estimativa do sinal *crosspolar*

Destes resultados conclui-se que para uma correcta estimativa de potência do sinal *crosspolar*, este deve ser no mínimo da mesma ordem de grandeza que o nível de ruído do sistema.

Estes resultados foram obtidos utilizando a mesma fonte de ruído nos canais copolar e crosspolar. Por este motivo não se espera o desempenho ideal pois o ruído encontra-se correlacionado nos dois canais. De modo a obter resultados realistas seria necessário que este fosse gerado em fontes independentes. Dado que estes testes sobre o receptor total se efectuam com sinais bastante fracos bastaria ter dois amplificadores com um ganho na ordem dos 30 a 40dB dedicados simplesmente a produzir ruído que seria adicionado a cada um dos canais.

6.3 Estimativa de fase relativa

Este é o teste prático mais complicado de realizar. Devido à frequência de funcionamento seria necessário a utilização de cabos suficientemente longos para impor um atraso de fase significativo entre o *copolar* e o *crosspolar* (na ordem de alguns metros). Por outro lado seria necessário saber o valor de permeabilidade e permitividade do cabo com alguma precisão para determinar o comprimento de onda para o cabo usado, e assim verificar a veracidade da estimativa. Por estes motivos o teste que aqui se apresenta mostra até que ponto a fase estimada se mantém estável à medida que o *copolar* e *crosspolar* se aproximam do nível de ruído.

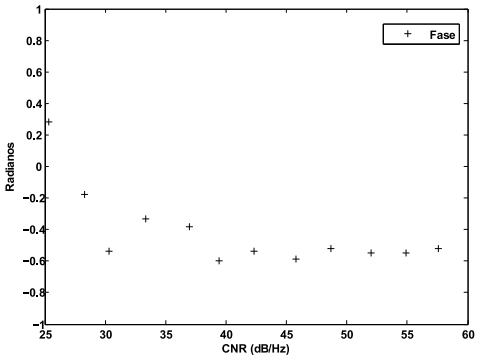


Figura 6.4: Teste de estabilidade da estimativa da fase relativa do sinal *crosspolar*

Os valores utilizados para o gráfico da figura 6.4 foram retirados durante o teste explicado no

tópico 6.3.

Do gráfico pode-se concluir que a estimativa da fase relativa não sofre alterações significativas à medida que a CNR se deteriora, tal como esperado do algoritmo explicado no tópico 3.3.3. Apesar de demonstrar ser consideravelmente estável observam-se algumas flutuações de maior amplitude para uma CNR inferior a aproximadamente 35dB. Tal como foi explicado, a precisão deste método depende em grande parte do número de amostras utilizado para efectuar a análise. Por este motivo, pode-se inferir que caso seja necessário deve utilizar um período de amostragem mais longo para reduzir a influência do ruído.

6.4 Seguimento

Foram realizados alguns testes simples para verificar o seguimento de sinal. Estes basearam-se no varrimento manual de frequência de um sinal proveniente de um gerador.

Verificou-se que nos casos em que a CNR apresentava os níveis esperados o seguimento era feito correctamente. Também se pode comprovar que qualquer alteração de frequência era ignorado quando a CNR se encontrava a baixo do limiar estabelecido.

No entanto seria interessante efectuar estes testes utilizando sinais mais realistas.

Capítulo 7

Conclusões Finais e Trabalho Futuro

7.1 Conclusões

Podemos concluir que o objectivo principal foi cumprido com sucesso, abrindo caminho para futuros melhoramentos.

Provou-se que é possível desenvolver um receptor digital de *beacon* com um baixo orçamento, recorrendo a soluções *open source* e *open hardware*, neste caso o GNU Radio em conjunto com o USRP.

Adicionalmente foi criada a base para a construção de um interface simples e eficiente com capacidade de receber novas funções ao seu reportório.

Verificou-se que o sistema apresenta um comportamento muito linear tanto na estimativa da amplitude como na determinação da frequência.

O processo de sintonia mostrou-se relativamente lento, no entanto os resultados em todas as ocasiões devolveu valores correctos para o início da aquisição. Durante a aquisição, os valores tanto do *copolar* como do *crosspolar*, apresentaram erros baixos em função do valor esperado. Também em termos de gama de medição foi possível atingir o patamar mínimo de 17dB de CNR, momento em que os valores estimados para a fase se tornaram completamente aleatórios. Em termos de registos, não houve qualquer problema na sua leitura mesmo quando ocorreram falhas no sistema, mostrando que nenhuma informação é perdida em situações limite.

A funcionalidade de *upload* para FTP demonstrou ter falhas quando o sistema tem uma conexão lenta. Suspeita-se tal se deva a que múltiplos pedidos sobrepostos e a uma falha no tratamento desta situação.

Após a conclusão de todos estes pontos, este projecto deixa claro o grande potencial do SDR assim como a ideia de que existe muito espaço para melhoramentos em termos de funcionalidade e comportamentos face a variações do sinal.

Em resumo, no final da realização desta dissertação foram adquiridos ou aprofundados conhecimentos nas seguintes áreas:

- Principais efeitos de perturbação da propagação de ondas electromagnéticas e como registá-los;
- Processamento digital de sinal;
- Programação em linguagem Python e C++;
- Utilização da plataforma GNU Radio em conjunto com o *hardware* USRP como meio de implementar um receptor digital;
- Métodos de seguimento de um sinal com deslize em frequência;

- Criação de interfaces, recorrendo a bibliotecas wxPython, com capacidade de gestão de tarefas em *multi-thread*;
- Criação de ficheiros de registo em tempo real compatíveis com MATLAB assim como a respectiva função de leitura.

7.2 Trabalho Futuro

Futuramente poderão ser feitas as seguintes alterações e ensaios:

- Captura de uma série temporal do sinal e *display* gráfico da mesma;
- Melhoramento do suporte a upload para FTP;
- Implementação de *software* de auxilio à calibração semi-automática;
- Estudo de outros algoritmos para as estimativas. Visto que os actuais consomem aproximadamente 20% do total de recursos, existe margem para introduzir maior complexidade;
- Algoritmo de seguimento de frequência baseado numa FLL;
- Testes mais completos de desempenho focando situações de baixa CNR;
- Ligação dos sensores da estação meteorológica ao USRP;
- Implementar o controlo de um conjunto de motores externos que alterem a posição da antena de modo a obter uma melhor apontamento da antena ao longo do dia.

Anexo A

Nota: todos os comandos são executados num terminal do sistema operativo Ubuntu.

1 - Adicionar repositórios à lista

Abrir um terminal e executar o comando seguinte:

```
sudo vim /etc/apt/sources.list
```

Adicionar as linhas seguintes ao ficheiro de texto aberto:

```
deb http://us.archive.ubuntu.com/ubuntu/ jaunty main restricted universe multiverse
deb http://us.archive.ubuntu.com/ubuntu/ jaunty-updates main restricted universe multiverse
deb http://security.ubuntu.com/ubuntu/ jaunty-security main restricted universe multiverse
```

Após guardar as alterações e fechar o ficheiro, actualizar *software* disponível nos repositórios:

```
sudo apt-get update
```

---

2 - Instalar ferramentas de *software* base:

Dentro do terminal executar as seguintes linhas:

```
sudo apt-get -y install swig g++ automake1.9 libtool python2.5-dev fftw3-dev \
libcppunit-dev libboost1.35-dev sdcc-nf libusb-dev \
libsdl1.2-dev python-wxgtk2.8 subversion guile-1.8-dev \
libqt4-dev python-numpy ccache python-opengl libgsl0-dev \
python-cheetah python-lxml doxygen qt4-dev-tools \
libqwt5-qt4-dev libqwtplot3d-qt4-dev pyqt4-dev-tools

sudo apt-get -y install gkrellm wx-common libwxgtk2.8-dev alsa-base \
autoconf xorg-dev gfortran gawk bison openssh-server cvs
```

---

3 - Instalar Boost C++ Libraries:

Esta biblioteca é essencial para a criação de ponteiros para as classes criadas pelas bibliotecas fornecidas pelo GNU Radio. Os ficheiros do projecto Boost estão alojados no servidor Sourceforge[23]. A versão utilizada foi a 1.38.0.

Copiar o ficheiro `boost_1_38_0.tar.gz` para um local temporário e executar as seguintes linhas no terminal:

```
tar xvvf boost_1_38_0.tar.gz
cd boost_1_38_0
BOOST_PREFIX=/opt/boost_1_38_0
./configure --prefix=$BOOST_PREFIX --with-libraries=thread,date_time,program_options
```

```
make
sudo make install
```

---

4 - Instalar Blibliotecas GNU Radio:

Primeiro é necessário descarregar a última versão estável do código. Só depois pode ser compilado e instalado. Para tal basta executar as linhas seguintes no terminal:

```
svn co http://gnuradio.org/svn/gnuradio/branches/releases/3.2 gnuradio
cd gnuradio
export LD_LIBRARY_PATH=$BOOST_PREFIX/lib
./bootstrap
./configure --with-boost=$BOOST_PREFIX
make
make check
sudo make install
```

---

5 - Adicionar direitos de acesso às bibliotecas:

Neste momento todo o *software* se encontra instalado e falta apenas adicionar os utilizadores que pertencem ao grupo de desenvolvimento à lista de utilizadores com acesso ao *hardware* USRP. Isto é necessário devido ao modo como o LINUX trata o reconhecimento de dispositivos com *hotplug*.
As seguintes linhas de comando darão permissões ao utilizador cujo nome é "user":

```
sudo addgroup usrp
sudo usermod -G usrp -a user
echo 'ACTION==``add'',BUS==``usb'',SYSFS{idVendor}==``fffe'',SYSFS{idProduct}==``0002'',
GROUP:=``usrp'',MODE:=``0660''' > tmpfile
sudo chown root.root tmpfile
sudo mv tmpfile /etc/udev/rules.d/10-usrp.rules

sudo /etc/init.d/udev stop
sudo /etc/init.d/udev start
```

---

6 - Indicar caminho no sistema das bibliotecas Boost C++:

Embora não seja obrigatório pode evitar problemas futuros durante as compilações.
As seguintes linhas devem ser executadas no terminal:

```
cp /etc/ld.so.conf /tmp/ld.so.conf
echo /usr/local/lib >> /tmp/ld.so.conf
echo /opt/boost_1_37_0/lib >> /tmp/ld.so.conf
sudo mv /tmp/ld.so.conf /etc/ld.so.conf
sudo ldconfig
```

---

7 - Verificar instalação:

É hora de confirmar que o nosso ambiente de programação e sistema USRP estão prontos para começar

o trabalho de desenvolvimento. Durante a instalação foram copiados alguns exemplos que provam ser úteis não só para aprender mas também para efectuar testes rápidos do sistema. Optámos por correr o exemplo `usrp_wfm_rcv.py`, que implementa o receptor de rádio FM descrito anteriormente.
Os seguintes comandos devem ser executados para efectuar a verificação do sistema:

```
cd gnuradio/gnuradio-examples/python/usrp
./usrp_wfm_rcv.py
```

Se tudo correu como esperado neste momento vemos uma janela semelhante à figura 4.9 onde podemos confirmar que o USRP está a receber sinal e efectuar uma leitura correcta.

Anexo B

Este módulo contém a função `reg_calc(F_OUT)` que permite reprogramar a PLL para sintetizar a frequência `F_OUT`.

Para utilizar este módulo basta incluir "`from reg_calc import reg_calc`" no código Python.

---

reg_calc.py

```
def reg_calc(F_OUT, gnu):

   def hex2str(num):
      """
      Takes the hexadecimal number num and converts it
      to a string
      """
      num_l =  num        & HEX_MSK
      num_m = (num >>  8) & HEX_MSK
      num_h = (num >> 16) & HEX_MSK
      return chr(num_h)+chr(num_m)+chr(num_l)

   u = gnu

   # Predefined values #
   HEX_MSK = 0xff
   MOD = 3698
   R = 6
   N_INT = 108

   # N_FRAC calculation #
   N_FRAC = int(MOD * ((R*F_OUT)/110.9373e6 - N_INT))

   # Noise Spur Reg 0 #
   NOISE0_REG = 0x000003
   spi_cmd = hex2str(NOISE0_REG)

   # Noise Spur Reg 1 #
   NOISE1_REG = 0x0003c7
   spi_cmd = hex2str(NOISE1_REG)

   # Control Reg 0 #
   CTRL0_REG = 0x0003c6
   spi_cmd = hex2str(CTRL0_REG)

   # Control Reg 1 #
   CTRL1_REG = 0x0003c2
   spi_cmd = hex2str(CTRL1_REG)

   # N DIVIDER REG #
   N_DIV_MSK = 0x7ffffc
   N_CTRL = 0x0
   N_REG = (N_INT << 14 ) | (N_FRAC << 2) | N_CTRL & N_DIV_MSK
   spi_cmd = hex2str(N_REG)
```

```
# R DIVIDER REG #
R_DIV_MSK = 0x07fffd
R_CTRL = 0x1
R_PRE = 0x1
R_REG = (R << 14) | (MOD << 2) | R_CTRL | (R_PRE << 18) & R_DIV_MSK
spi_cmd = hex2str(R_REG)
```

Anexo C

C.1 Classe TunerFlowGraph()

Código em detalhe da classe `TunerFlowGraph()`.

tuning.py

```
class TunerFlowGraph(gr.top_block):
  def __init__(self, CenterFrequency=10.7e6, FFTSize=32768):
    gr.top_block.__init__(self)

    # Set frequency and decimation filters
    decimation = 256
    SamplingFrequency = int(64e6/decimation);
    sw_decim = 5
    chan_filt_decim = sw_decim
    chan_filt_coeffs = gr.firdes.low_pass (1,        # gain
                            SamplingFrequency,      # sampling rate
                            10000,                  # midpoint of transition band
                            20000,                  # width of transition band
                            gr.firdes.WIN_BLACKMAN) # filter type
    chan_filt = gr.fir_filter_ccf (chan_filt_decim, chan_filt_coeffs)

    # Set and tune Co-Polar Channel
    self.u = usrp.source_c(0, decimation, fpga_filename="std_2rxhb_2tx.rbf")
    self.u.set_nchannels(1)
    self.subdev_spec = (0,0)
    self.subdev = usrp.selected_subdev(self.u, self.subdev_spec)
    self.subdev.set_auto_tr(True)
    self.subdev.select_rx_antenna('RX2')
    self.u.tune(0, self.subdev, CenterFrequency)
    g = self.subdev.gain_range()
    gain = float(g[0]+g[1])/2
    self.subdev.set_gain(gain)

    # Block Creation
    # FFT
    mywin = window.blackmanharris(FFTSize)
    fft = gr.fft_vcc(FFTSize, True, mywin)
    # Complex to Magnitude
    c2m = gr.complex_to_mag(FFTSize)
    # Index of the Maximum Value
    imax = gr.argmax_fs(FFTSize)
    # Amplitude of the Maximum Value
    amax = gr.max_ff(FFTSize)
    # Short to Float
    s2f1 = gr.short_to_float()
    s2f2 = gr.short_to_float()
    # Stream to Vector
    ss2v = gr.stream_to_vector(gr.sizeof_gr_complex, FFTSize)
    # Vector to Stream
    v2ss = gr.vector_to_stream(gr.sizeof_float, FFTSize)
    # Probe
    probe = gr.probe_signal_f()
    # Buffers
```

```
    self.buffer_freq = howto.buffer_sink(2000, CenterFrequency,
                                         FFTSize, SamplingFrequency/sw_decim)
    self.buffer_amp = howto.buffer_sink(2000, CenterFrequency,
                                        FFTSize, SamplingFrequency/sw_decim)
    self.buffer_fft = howto.buffer_sink(FFTSize, CenterFrequency,
                                        FFTSize, SamplingFrequency/sw_decim)

    # Connections
    self.connect(self.u, chan_filt,  ss2v, fft, c2m, (imax, 0), s2f1, self.buffer_freq)
    self.connect((imax, 1), s2f2, probe)
    self.connect(c2m, amax, self.buffer_amp)
    self.connect(c2m, v2ss, self.buffer_fft)

    # PLL configuration @ 2010.7MHz defeault
self.F_PLL = 2.0107e9
reg_calc(self.F_PLL,self.u)
```

C.2 Classe Tuner()

Código em detalhe da classe `Tuner()`.

---

tuning.py

```
class Tuner():
  def __init__(self, frame):
    self.frame = frame
    self.Stop = False

    # Start tuning process
    def StartTuner (self):
      # Set control values to fine tune the signal frequency
      self.thld = -20
      cf = 10.7e6
      cf_threshold = (cf-3, cf+3)
      fs = (64e6/256)/5
      fsize = 32768
      bw = ((cf-fs/2)/1e6, (cf+fs/2)/1e6)
      bin = fs/fsize
      qf = 0

      # Start flowgraph
      tb = TunerFlowGraph(cf, fsize)
      tb.start()
      time.sleep(4)

      # Tuning data average
      mg = tb.buffer_amp.out_mag()[-5:]
      mg_avg = sum(mg)/len(mg)
      fq = tb.buffer_freq.out_freq()[-5:]
      fq_avg = sum(fq)/len(fq)

      # Evaluate signal quality
      if mg_avg > self.thld:
        if fq_avg > cf_threshold[0]:
          if fq_avg < cf_threshold[1]:
            count += 1
          else:
            cf = fq_avg
            cf_threshold = (cf-3, cf+3)
            bw = ((cf-fs/2)/1e6, (cf+fs/2)/1e6)
            count = 0
            tb.buffer_freq.SetFrequency(cf)
```

```
      tb.buffer_amp.SetFrequency(cf)
      tb.buffer_fft.SetFrequency(cf)
      tb.u.tune(0, tb.subdev, cf)
    else:
      cf = fq_avg
      cf_threshold = (cf-3, cf+3)
      bw = ((cf-fs/2)/1e6, (cf+fs/2)/1e6)
      count = 0
      tb.buffer_freq.SetFrequency(cf)
      tb.buffer_amp.SetFrequency(cf)
      tb.buffer_fft.SetFrequency(cf)
      tb.u.tune(0, tb.subdev, cf)
elif mg_avg <= self.thld:
  count = 0

# Send data to main thread
data = []
x = [(i*bin+(cf-fs/2))/1e6 for i in range(fsize)]
y = tb.buffer_fft.out_fft()
for n in range(len(x)):
  data.append((x[n], y[n]))
  data_line = wx.lib.plot.PolyLine(data, colour='blue', width=1)
  gc = wx.lib.plot.PlotGraphics([data_line],"CoPolar", "Mhz", "dB")

  stats = {}
  stats["Step"] = 1
  stats["Frequency (Hz)"] = cf
  stats["Sampling (Hz)"] = fs
  stats["FFT Size (bins)"] = fsize
  stats["Quality (0-5)"] = count
  stats["Amplitude (dB)"] = mg_avg

  wx.PostEvent(self.frame, events.TunerResultEvent((gc, bw, stats)))

# Enter tuning loop after first estimation
step=2
while(True):
  if not self.Stop:
    time.sleep(4)

    # Tuning data average
    mg = tb.buffer_amp.out_mag()[-5:]
    mg_avg = sum(mg)/len(mg)
    fq = tb.buffer_freq.out_freq()[-5:]
    fq_avg = sum(fq)/len(fq)

    # Evaluate signal quality
    if mg_avg > self.thld:
      if fq_avg > cf_threshold[0]:
        if fq_avg < cf_threshold[1]:
          count += 1
          if count >= 5:
            self.frame.NCOfrequency = cf
            self.Stop = True
        else:
          cf = fq_avg
          cf_threshold = (cf-3, cf+3)
          bw = ((cf-fs/2)/1e6, (cf+fs/2)/1e6)
          count = 0
          tb.buffer_freq.SetFrequency(cf)
          tb.buffer_amp.SetFrequency(cf)
          tb.buffer_fft.SetFrequency(cf)
          tb.u.tune(0, tb.subdev, cf)
      else:
        cf = fq_avg
        cf_threshold = (cf-3, cf+3)
        bw = ((cf-fs/2)/1e6, (cf+fs/2)/1e6)
        count = 0
        tb.buffer_freq.SetFrequency(cf)
        tb.buffer_amp.SetFrequency(cf)
```

```
            tb.buffer_fft.SetFrequency(cf)
            tb.u.tune(0, tb.subdev, cf)
        elif mg_avg <= self.thld:
            count = 0

        # Send data to main thread
        data = []
        x = [(i*bin+(cf-fs/2))/1e6 for i in range(fsize)]
        y = tb.buffer_fft.out_fft()
        for n in range(len(x)):
          data.append((x[n], y[n]))
          data_line = wx.lib.plot.PolyLine(data, colour='blue', width=1)
          gc = wx.lib.plot.PlotGraphics([data_line],"CoPolar", "Mhz", "dB")

          stats = {}
          stats["Step"] = step
          stats["Frequency (Hz)"] = cf
          stats["Sampling (Hz)"] = fs
          stats["FFT Size (bins)"] = fsize
          stats["Quality (0-5)"] = count
          stats["Amplitude (dB)"] = mg_avg

        wx.PostEvent(self.frame, events.TunerResultEvent((gc, bw, stats)))
        step += 1

        # If the user forced the loop to stop
        if self.Stop:
          tb.stop()
          tb = None
          break

# Allows the tuning loop to be stoped from the outside
def StopTuner(self):
  self.Stop = True
```

Anexo D

Este anexo apresenta os seguintes eventos:

- `StatusEvent()`: actualiza a barra de estado da GUI;
- `TunerResultEvent()`: envia os dados da *thread* de sintonia para a GUI;
- `AcquisitionResultEvent()`: envia os dados da *thread* de aquisição para a GUI;
- `TuningEnablerEvent()`: comuta o estado da janela de sintonia entre visível e invisível.

Todos estes eventos têm que ser chamados pela *thread* que gera a informação e atendidos pela *thread* principal que gere a GUI.

---

events.py

```
import wx

# Status bar update #
EVT_STATUS_ID = wx.NewId()

def EVT_STATUS(win, func):
  win.Connect(-1, -1, EVT_STATUS_ID, func)

class StatusEvent(wx.PyEvent):
  def __init__(self, status):
    wx.PyEvent.__init__(self)
    self.SetEventType(EVT_STATUS_ID)
    self.status = status

# Return results from tuning #
EVT_TUNER_RESULT_ID = wx.NewId()

def EVT_TUNER_RESULT(win, func):
  win.Connect(-1, -1, EVT_TUNER_RESULT_ID, func)

class TunerResultEvent(wx.PyEvent):
  def __init__(self, data):
    wx.PyEvent.__init__(self)
    self.SetEventType(EVT_TUNER_RESULT_ID)
    self.data = data

# Return results from aquisiton #
EVT_ACQUISITION_RESULT_ID = wx.NewId()

def EVT_ACQUISITION_RESULT(win, func):
  win.Connect(-1, -1, EVT_ACQUISITION_RESULT_ID, func)

class AcquisitionResultEvent(wx.PyEvent):
  def __init__(self, data):
    wx.PyEvent.__init__(self)
    self.SetEventType(EVT_ACQUISITION_RESULT_ID)
```

```
    self.data = data

# Shows/Hides tuning window #
EVT_TUNING_WIN_ENABLER_ID = wx.NewId()

def EVT_TUNING_WIN_ENABLER(win, func):
    win.Connect(-1, -1, EVT_TUNING_WIN_ENABLER_ID, func)

class TuningEnablerEvent(wx.PyEvent):
  def __init__(self, data):
    wx.PyEvent.__init__(self)
    self.SetEventType(EVT_TUNING_WIN_ENABLER_ID)
    self.data = data
```

Bibliografia

[1] *All Digital Transceiver ADT-200A*. http://www.adat.ch/index_e.html, Acedido em Outubro de 2009.

[2] *Apple Mac OSX*. http://www.apple.com/macosx/, Acedido em Outubro de 2009.

[3] *ATI Stream Technology*. http://www.amd.com/us/products/technologies/stream-technology/Pages/stream-technology.aspx, Acedido em Outubro de 2009.

[4] *CentOS - The Community ENTerprise Operating System*. http://www.centos.org/, Acedido em Outubro de 2009.

[5] *cplusplus.com - The C++ Resources Network*. http://www.cplusplus.com, Acedido em Outubro de 2009.

[6] *Debian*. http://www.debian.org/, Acedido em Outubro de 2009.

[7] *ESA Telecomunication & Integrated Aplications - AlphaSat*. http://telecom.esa.int/telecom/www/object/index.cfm?fobjectid=1138, Acedido em Outubro de 2009.

[8] *Ettus Research*. http://www.ettus.com/, Acedido em Outubro de 2009.

[9] *fedora*. http://fedoraproject.org/pt/, Acedido em Outubro de 2009.

[10] *GNU + Cygnus + Windows = Cygwin*. http://www.cygwin.com/, Acedido em Outubro de 2009.

[11] *GNU Radio*. http://gnuradio.org/trac, Acedido em Julho de 2009.

[12] *GNU Radio Forum*. http://www.nabble.com/GnuRadio-f1878.html, Acedido em Outubro de 2009.

[13] *GNU Radio Ubuntu Installation Guide*. http://gnuradio.org/trac/wiki/UbuntuInstall, Acedido em Outubro de 2009.

[14] *How to Write a Signal Processing Block*. http://www.rfspace.com/SDR-IQ.html, Acedido em Outubro de 2009.

[15] *How to Write a Signal Processing Block*. http://www.gnu.org/software/gnuradio/doc/howto-write-a-block.html, Acedido em Outubro de 2009.

[16] *MathWorks MATLAB*. http://www.mathworks.com/products/matlab/, Acedido em Outubro de 2009.

[17] *Microtelecom Perseus*. http://www.microtelecom.it/perseus/, Acedido em Outubro de 2009.

[18] *MinGW - Minimalistic GNU for Windows*. http://www.mingw.org/, Acedido em Outubro de 2009.

[19] *The NetBSD Project*. http://www.netbsd.org/, Acedido em Outubro de 2009.

[20] *nVidia Cuda Zone*. http://www.nvidia.com/object/cuda_home.html#, Acedido em Outubro de 2009.

[21] *openSUSE*. http://pt.opensuse.org/Bem-vindo_ao_openSUSE.org, Acedido em Outubro de 2009.

[22] *Python Programming Language – Official Website*. http://www.python.org/, Acedido em Outubro de 2009.

[23] *Sourceforge - Boost C++ Libraries*. http://sourceforge.net/projects/boost/files/, Acedido em Outubro de 2009.

[24] *SRL QuickSilver QS1R VERB*. http://www.srl-llc.com/, Acedido em Outubro de 2009.

[25] *ubuntu*. http://www.ubuntu.com/, Acedido em Outubro de 2009.

[26] *wxPython*. http://www.wxpython.org/, Acedido em Outubro de 2009.

[27] *wxPython*. http://qt.nokia.com/products/, Acedido em Outubro de 2009.

[28] E. M. Cabral. *Unidade Interior de um Receptor de Satélites Baseado em Kit GNU Rádio*. Tese de Mestrado em Engenharia Electrónica e Telecomunicações(submetida), November 2009.

[29] K. Cornelis, B. B. Thorpe, and O. Teong. *A DSP Based Satellite Beacon Receiver and Radiometer*. 1998.

[30] K. Davies. *Recent progress in satellite radio beacon studies with particular emphasis on the ATS-6 radio beacon experiment*, volume 25. Springer Netherlands, April 1980.

[31] R. L. M. de Sousa. *Receptor Digital para Medição de Balizas de Satélite*. 2007.

[32] A. Devices. *Fractional-N Frequency Synthesizer ADF4153*. 2008.

[33] A. Devices. *MT-085 Tutorial: Fundamentals of Direct Digital Synthesis (DDS)*. 2009.

[34] O. K. Ersoy. *A Comparative Review of Real and Complex Fourier-Related Transforms*, volume 82. IEEE Xplore, March 2008.

[35] E. European Space Agency. *AlphaSat Fact Sheet*. January 2009.

[36] M. I. Frederic J. Harris. *On then Use of Windows for Harmonic Analysis with the Discrete Fourier Transform*. January 1978.

[37] F. A. Hamza. *The USRP under 1.5X Magnifying Lens!* June 2008.

[38] N. Instruments™. *The Fundamentals of FFT-Based Signal Analysis and Measurement in LabVIEW and LabWindows/CVI*. 2006.

[39] I. T. U. International Telecommunications Union. *Handbook on Satellite Communications*. Wiley-Interscience, 3rd edition, April 2002.

[40] J. S. Mandeep. *Microwave depolarization versus rain attenuation on earth space in Malaysia*, volume 6. 2008.

[41] J. S. Mandeep and S. I. S. Hassan. *Cloud Attenuation in Millimeter Wave and Microwave Frequencies for Satellite Applications over Equatorial Climate*, volume 29. Springer New York, February 2008.

[42] MAXIM. *The ABCs of ADCs: Understanding How ADC Errors Affect System Performance*. `http://www.maxim-ic.com/appnotes.cfm/an_pk/748/`, Acedido em Julho de 2009.

[43] P. G. Pino, J. M. Garcia, and A. Benarroch. *Tropospheric scintillation measurements on a Ka-Band satellite link in Madrid*. 2008.

[44] N. Rappin and R. Dunn. *WxPython in action*. Manning, Greenwich, Conn, 2006.